UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Autenticación descentralizada mediante criptografía asimétrica
Autor: Raúl Benencia
Director: Lic. Javier Díaz
Carrera: Licenciatura en Informática

Resumen

El presente trabajo de grado describe, diseña e implementa un sistema de autenticación con énfasis en la descentralización de credenciales. Para tal fin, se realiza un análisis del estado del arte de las tecnologías de autenticación vía Internet con estándares abiertos, se identifican diversas debilidades en el paradigma y se solucionan con el mecanismo propuesto.

El trabajo plantea que la descentralización de las credenciales se puede lograr utilizando criptografía asimétrica. Para ello se asignan uno o más juegos de claves a cada usuario, se almacenan las claves públicas del lado del servidor y se proporciona un mecanismo para que el usuario conserve las respectivas claves privadas en sus propios dispositivos.

Finalmente, con el objetivo de eliminar la limitación de poder autenticarse únicamente desde los dispositivos que posean una clave privada asociada, se propuso un mecanismo que realiza el proceso de autenticación utilizando la cámara de un dispositivo móvil del usuario para interpretar códigos QR y descifrar su contenido.

Palabras Claves

autenticación, descentralización, criptografía asimétrica, RSA, OpenID, OAuth, OpenID Connect, aplicación web, *websockets,* software libre, GPL, programación funcional, Haskell

Trabajos Realizados

Se diseñó un sistema de autenticación cuya principal característica es la descentralización de las credenciales entre sus usuarios. Se propuso una posible arquitectura de la infraestructura del sistema de autenticación junto con un protocolo de comunicación entre quien autentica y quien debe ser autenticado. Finalmente, se implementó una prueba de concepto funcional que utiliza las ideas desarrolladas en la tesina.

Conclusiones

Se ha desarrollado un mecanismo de autenticación que no centraliza credenciales, fomenta el uso de claves distintas para sitios distintos y, por último, no depende de una entidad externa para realizar el proceso de autenticación.
Se reconoce que la adopción masiva de este sistema se dificulta debido la complejidad inherente de requerir un software adicional que corra del lado del cliente. Sin embargo, existen ambientes de trabajo donde su utilización es propicia.

Trabajos Futuros

Como líneas de investigación a futuro se propone:

- Implementar módulos de autenticación en diversas plataformas web.
- Brindar soporte a distintos tipos de algoritmos de criptografía asimétrica.
- Extender la automatización de la asociación de cuentas entre el sistema que se ejecuta localmente y los sitios que utilizan el mecanismo de autenticación propuesto.

Fecha de la presentación: Junio 2014

UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

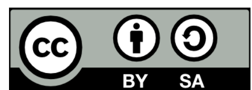
Autenticación descentralizada mediante criptografía asimétrica

Alumno:
Raúl BENENCIA

Director:
Lic. Javier DÍAZ

Junio de 2014

Esta obra fue realizada en su totalidad utilizando software libre y se encuentra bajo licencia Creative Commons Atribución-Compartir Igual 4.0 Internacional.



Resumen

El presente trabajo de grado describe los principales métodos de autenticación actuales en el ámbito de Internet, analiza sus fortalezas y debilidades y finalmente ofrece un método de autenticación alternativo que pretende solucionar las principales desventajas de los métodos analizados.

A mis viejos...

Índice general

1. Introducción 5
1.1. Motivación 5
1.2. Objetivos 6
1.3. Contribuciones 7

2. Estado del arte 8
2.1. Tecnologías actuales 8
2.1.1. OpenID, autenticación federada 8
2.1.2. OAuth, un *framework* de autorización 12
2.1.3. OpenID Connect y *logins* sociales 15
2.2. Problemas actuales en la autenticación 16
2.2.1. Centralización de credenciales 16
2.2.2. Homogeneidad de credenciales 17
2.2.3. Entidad de autenticación externa 18

3. Solución propuesta 20
3.1. Problemática 20
3.1.1. Centralización 21
3.1.2. Homogeneidad 23
3.1.3. Entidad externa 24
3.2. Infraestructura 25
3.2.1. Depósito de claves 25
3.2.2. Manejador de *challenges* 26
3.2.3. Módulos de autenticación 27
3.2.4. Protocolo 28
3.2.5. Servidor 29
3.3. Autenticación móvil 31

4. Implementación y resultados 33
4.1. Servicio local 34
4.1.1. *Frontend* de la aplicación 34
4.1.2. *Backend* de la aplicación 40
4.1.3. Sobre la elección del lenguaje 45
4.1.4. El *framework* Yesod 47
4.2. Servidor de ejemplo 49
4.2.1. Funcionalidad de la aplicación 49
4.2.2. Sobre del desarrollo de la aplicación 51
4.3. Aplicación para dispositivos móviles 51

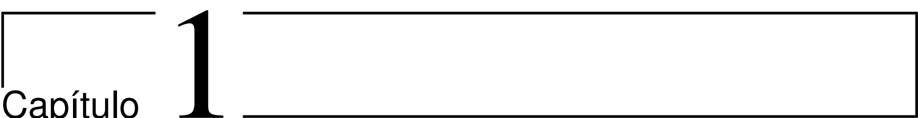
5. Conclusión 54
5.1. Trabajo futuro 57

Índice de figuras

2.1. Estadísticas de uso de *logins* sociales 15

3.1. Intraestructura 25
3.2. Depósito de claves 26
3.3. Manejador de *challenges* 28

4.1. Captura de pantalla del *Dashboard* 35
4.2. Captura de pantalla del módulo **Claves** 36
4.3. Captura de pantalla de un juego de claves exportado 37
4.4. Capturas de pantalla con información acerca de un juego de claves 37
4.5. Creación de una nueva clave asimétrica 38
4.6. Listado de cuentas asociadas al sistema 39
4.7. Creación de una nueva cuenta 39
4.8. Consulta en la tabla `account` 40
4.9. Consulta en la tabla `key_pair` 41
4.10. Consulta en la tabla `key_account` 41
4.11. Consulta en la tabla `log` 42
4.12. Ejemplo de implementación del protocolo de comunicación . . 44
4.13. El logotipo del lenguaje Haskell 46
4.14. Configuración de rutas en Yesod 47
4.15. *Benchmarks* de aplicaciones web 48
4.16. Captura de pantalla de la página inicial 49
4.17. Captura de pantalla de la página de registro 50
4.18. Captura de pantalla del código QR de un *challenge* 51
4.19. Menú principal de la aplicación para teléfonos 52



Introducción

El presente trabajo de grado plantea la posibilidad de utilizar un mecanismo de autenticación cuya principal característica sea que las credenciales no estén centralizadas en una entidad externa. Por el contrario, y a diferencia de los mecanismos actuales más utilizados, las credenciales serán almacenadas por los usuarios en sus propios dispositivos.

Con el objetivo de demostrar el funcionamiento del mecanismo de autenticación se desarrolló una implementación de la propuesta que pone en práctica los conceptos expuestos en el siguiente capítulo. De esta forma se demuestra que la utilización a gran escala de un sistema con estas características es posible.

1.1. Motivación

Sin duda el mecanismo de autenticación más utilizado hoy en día es un nombre de usuario junto con su contraseña. Poco a poco se está empezando a tomar conciencia de que no deben utilizarse las mismas contraseñas para distintos sitios, y se sugiere a los usuarios que utilicen contraseñas distintas en cada nueva cuenta que adquieran. Esto se debe principalmente a que, si un atacante compromete la base de datos de usuarios que posee un sitio con seguridad débil y, por ejemplo, las contraseñas se guardan en texto plano, entonces dicho atacante dispondrá de acceso a todas las cuentas del usuario que utilicen la misma contraseña.

Utilizar contraseñas distintas (y recordarlas) pierde escalabilidad a medida que aumenta el número de cuentas del usuario. Para solucionar esta desventaja, la

tendencia actual es utilizar *Single Sign-On*, delegando la responsabilidad de autenticación a una única entidad, y por lo tanto reduciendo la necesidad de recordar muchas contraseñas.

Sin embargo, delegar la responsabilidad de autenticación a unas pocas entidades es cuestionable desde el punto de vista de la privacidad del usuario, pues inevitablemente el proveedor de *Single Sign-On* conocerá todos los sitios a los que accede, la hora a la que accede a ellos, etc. Teniendo en cuenta los actuales casos de espionaje por parte de gobiernos extranjeros con información brindada por importantes proveedores de *Single Sign-On*, es menester plantear soluciones de autenticación alternativas.

1.2. Objetivos

El objetivo principal del presente trabajo será desarrollar un protocolo de autenticación mediante criptografía asimétrica. El protocolo comprenderá los procedimientos para la realización de consultas por claves públicas, la creación de *challenges* y todo el comportamiento necesario para lograr la autenticación del usuario.

También se desarrollará una implementación sobre HTTP[9] del protocolo propuesto, con el objetivo de que pueda ser utilizado para autenticar usuarios en aplicaciones web. Sin embargo, se pretende conseguir que el protocolo de autenticación sea independiente de la plataforma donde corre.

Se espera que el protocolo diseñado pueda ser fácilmente integrado en sistemas que utilicen una arquitectura de tipo cliente-servidor. Además, se espera conseguir que el mecanismo propuesto simule la capacidad de los sistemas *Single Sign-On* de poder ingresar una única contraseña para acceder a cuentas de usuario en distintos sistemas. En este caso, la contraseña ingresada por el usuario se utilizaría para desencriptar un archivo que cumple la función de depósito de claves privadas. Luego de desencriptar este archivo, el usuario podrá acceder libremente a todas las cuentas con claves públicas asociadas. En consecuencia, los usuarios de este mecanismo dispondrán de la principal ventaja de un sistema *Single Sign-On* sin por ello sacrificar su privacidad.

1.3. Contribuciones

El presente trabajo genera contribuciones, principalmente, de dos formas distintas. Por un lado hay una contribución intelectual, una idea que ha sido escrita de forma detallada. Por otro lado, hay una contribución material, que es todo el software escrito. Estas contribuciones juntas aportan un nuevo mecanismo de autenticación a los ya existentes en el ámbito de la autenticación vía Internet. Sin embargo, a diferencia de la gran cantidad de mecanismos existentes, este nuevo sistema tiene la particularidad de no centralizar las credenciales de autenticación en un servidor externo al usuario, lo cual proporciona grandes beneficios respecto a la privacidad del usuario.

La contribución intelectual es un protocolo que define cómo utilizar el mecanismo de autenticación propuesto. El protocolo explica cómo iniciar la comunicación entre las partes, cómo intercambiar credenciales y *challenges* y cómo determinar si la autenticación es o no correcta. Este protocolo es independiente de la tecnología subyacente, por lo cual se ha puesto empeño en que sea lo más genérico posible. De esta forma, el mecanismo propuesto podrá ser utilizado en protocolos tan diversos como se quiera, como HTTP o SMTP.

Por otro lado, la contribución material será un sistema de software que implemente el mencionado protocolo junto con el mecanismo propuesto. Como es de esperar en un sistema con estas características, el sistema de software está compuesto por varias partes que se ejecutan de forma independiente. Se ha escrito software que corre del lado del servidor y software que corre del lado del cliente. Se ha incluido un componente extra que elimina la limitación de precisar un único dispositivo para realizar la autenticación. Todos los componentes del sistema de software implementado han sido liberados bajo la licencia GPL versión 3, por lo cual el código genera un aporte a la comunidad de usuarios de software libre, permitiendo que se use, estudie, modifique y distribuya libremente.

Finalmente, con el objetivo de que implementar el mecanismo de autenticación sea lo más ameno posible, se han empaquetado las principales bibliotecas de software de las cuales depende el proyecto para que su instalación resulte sencilla en las principales distribuciones de GNU/Linux.

Estado del arte

Este capítulo analizará las tecnologías de autenticación y autorización actuales con estándares abiertos en el ámbito de Internet. En particular, se analizarán tecnologías que permitan hacer uso de la propiedad de *Single Sign-On*. Para cada tecnología se describirán sus principales características y su grado de utilización actual. Finalmente, se expondrán los problemas que se detectaron en el paradigma de autenticación actual.

2.1. Tecnologías actuales

El estado del arte en la autenticación y autorización por Internet mediante mecanismos con estándares abiertos y de libre utilización lo componen principalmente tres tecnologías: OpenID para realizar autenticación, OAuth para realizar autorización, y OpenID Connect, que unifica las primeras dos.

2.1.1. OpenID, autenticación federada

OpenID[32] es un estándar abierto que permite a sitios que implementen este tipo de autenticación, llamados *relying parties*, autenticar usuarios mediante entidades externas, llamados proveedores de identidad. Esto permite a los usuarios manejar sus identidades digitales y libera a los administradores de la necesidad de mantener un sistema de autenticación propio. El estándar de OpenID provee un *framework* que define cómo debe ser la comunicación entre una *relying party* y un proveedor de identidad.

La autenticacón OpenID provee un método para probar que un usuario es dueño de un *identificador*. Este identificador se asocia a las cuentas de usuario en las *relying parties*. Además, este procedimiento se realiza sin que la *relying party* conozca cuáles son las credenciales para acceder al identificador OpenID del usuario, como la contraseña, la dirección de email, u otra información confidencial.

OpenID fue diseñado para ser descentralizado, permitiendo al usuario elegir entre diversos proveedores de OpenID, o incluso utilizar uno propio. No hay ninguna autoridad central que deba aprobar el registro de una nueva *relying party* o proveedor de OpenID. El usuario podrá crear una cuenta en el proveedor que elija y utilizar su identificador en cualquier sitio que soporte este tipo de autenticación. El usuario podrá también, si así lo quiere, migrar su identificador entre los distintos proveedores de OpenID.

La autenticación OpenID utiliza únicamente requerimientos y respuestas HTTP(S) y, por lo tanto, no requiere que el usuario instale software adicional para utilizarlo. Para utilizar este tipo de autenticación se necesita únicamente un navegador web. Este mecanismo tampoco necesita hacer uso de *cookies* ni de ningún otro mecanismo de manejo de sesiones. Tampoco se precisa utilizar Javascript, lo que lo hace ideal para navegadores antiguos o ambientes donde la ejecución de este tipo de código no esté permitida, aunque si se lo utilizara se podría realizar todo el proceso de autenticación, vía requerimientos AJAX, sin que el usuario abandone la página que está visitando.

Elementos del protocolo

El estándar de autenticación de OpenID define varios elementos necesarios para realizar la autenticación.

- *Identifier*: este elemento representa un identificador, que consiste normalmente en una URI, incluido el *string* del protocolo (`http` o `https`), aunque también se pueden utilizar XRIs[33]. El estándar de OpenID define varios tipos de identificadores que serán mencionados y explicados a continuación.
- *User-Agent*: este elemento representa simplemente al navegador web del usuario final, aunque puede ser cualquier programa que implemente el protocolo HTTP 1.1.
- *Relying Party*: este elemento representa a la aplicación web que quiere autenticar un usuario. En otras palabras, representa aquella entidad que

precisa pruebas de que un identificador pertenece a un usuario.

- *Proveedor de OpenID*: este elemento representa a la entidad que se encarga de autorizar y validar que un identificador pertenezca a un usuario. También permite que nuevos usuarios se registren y adquieran sus identificadores.
- *OP Endpoint URL*: este elemento representa la URL que debe proporcionar un proveedor de OpenID para "conversar" utilizando los mensajes del protocolo de autenticación.
- *OP Identifier*: este elemento es un identificador que representa al proveedor de OpenID.
- *User-Supplied Identifier*: este elemento es un identificador provisto por el usuario a una *relying party*. Cuando se inicia el protocolo de autenticación, el usuario final podrá optar por proporcionar su propio identificador, o el identificador del proveedor de OpenID. Si utiliza este último, entonces el proveedor deberá asistir al usuario para que seleccione un identificador para ser enviado a la *relying party*.
- *Claimed Identifier*: este elemento es el identificador final que posee un usuario, y sobre el cual las *relying parties* pedirán pruebas al proveedor de OpenID de que pertenece a éste.
- *OP-Local Identifier*: este elemento es un identificador local que asigna el proveedor de OpenID al usuario.

Descripción del protocolo

El funcionamiento general del protocolo se describe en siete pasos:

1. El usuario final comienza el proceso de autenticación presentando su *User-Supplied Identifier* a la *Relying Party* mediante su *User-Agent*.
2. Luego de recibir el *User-Supplied Identifier*, la *Relying Party* se encargará de averiguar cuál es la *Endpoint URL* del proveedor de OpenID que utilizó el usuario.
3. Opcionalmente, la *Relying Party* y el proveedor de OpenID podrán optar por establecer una *asociación*, terminología utilizada por el estándar de OpenID para referirse a un secreto compartido. Luego, el proveedor de OpenID podrá utilizar esta asociación para firmar digitalmente los subsecuentes mensajes, y la *Relying Party* podrá verificarlos utilizando el mismo secreto.

4. La *Relying Party* redirigirá el *User-Agent* del usuario hacia el proveedor de OpenID requiriendo la autenticación del identificador del usuario.
5. El proveedor de OpenID determinará si el usuario final está autorizado a utilizar el identificador que envió a la *relying party*. La forma de hacer esto suele variar, pero en general se utiliza un usuario con una contraseña.
6. El proveedor de OpenID redirigirá al usuario hacia la *relying party*, notificándole si el usuario logró o no ser autorizado.
7. Finalmente, la *relying party* verificará que la información recibida sea correcta, y si el usuario logró verificar que el identificador presentado le pertenece, entonces ésta podrá autenticarlo.

Historia y uso

El protocolo inicial de OpenID fue propuesto en mayo del 2005 por Brad Fitzpatrick, creador de LiveJournal. Inicialmente el proyecto se llamaba Yadis, acrónimo de *Yet Another Distributed Identity System* (aún otro sistema de identidad distribuida), pero fue renombrado a OpenID cuando el dominio `openid.net` fue transferido a Six Apart, la empresa atrás de LiveJournal. Como es de esperar, al poco tiempo LiveJournal habilitó este sistema de autenticación para el uso de sus usuarios.

Durante los siguientes meses el proyecto tomó impulso y varias empresas propusieron extensiones e integraron sus soluciones existentes. Durante comienzos del año 2007 Symantec, Microsoft y otras empresas integraron OpenID en sus productos. En diciembre del mismo año se publicó la especificación de OpenID Authentication 2.0.

En enero de 2008, Yahoo! comenzó a soportar OpenID 2.0 como proveedor y *relying party*. En mayo, SourceForge hizo lo mismo. En julio, MySpace comenzó a brindar soporte únicamente como proveedor. En octubre Google lo imitó. Diversas empresas hicieron lo mismo, e incluso Facebook, en el año 2009, brindó soporte como *relying party*.

Actualmente el protocolo tradicional de OpenID está en declive en pos de una nueva tecnología, OpenID Connect[34], que debido a que usa OAuth como base, será explicada luego de introducir dicha tecnología.

2.1.2. OAuth, un *framework* de autorización

Las aplicaciones de hoy día tienden a utilizar información del usuario disponible en otros servicios con el fin de ofrecerle una mejor usabilidad. Por ejemplo, una red social podría solicitar acceso al correo electrónico de un usuario con el fin de realizar *data mining* sobre los correos y extraer así las direcciones de los contactos más frecuentes con los que se comunica. De esta forma, la red social podrá realizar recomendaciones al usuario sobre qué personas agregar, y así aumentar su grafo de relaciones. Sin embargo, al proporcionar las credenciales a la red social, ésta tendrá control completo sobre la cuenta de correo, y nada le impedirá, por ejemplo, que envíe correos a todos los contactos del usuario realizando publicidad sobre sus productos.

Surge la necesidad, entonces, de brindar a aplicaciones externas acceso a ciertos aspectos de una aplicación, y restringir el uso del resto de la funcionalidad. Siguiendo con el ejemplo del correo electrónico, a la red social se le debería permitir únicamente el acceso a la libreta de direcciones del usuario, mientras que se debería restringir el uso del resto de la funcionalidad, como enviar o recibir correos. Bajo este precepto nace OAuth[17][20][18].

OAuth permite a los usuarios de un servicio otorgar a una entidad externa acceso a sus recursos sin tener que proporcionar su contraseña. También provee una forma de otorgar acceso limitado a los recursos de la aplicación. Los recursos se pueden limitar en cuanto al alcance o duración, entre otros.

Dueño del recurso

OAuth modifica la típica arquitectura **cliente-servidor** para añadir un tercer componente, el **dueño del recurso**. El dueño del recurso es quien autoriza al cliente a utilizar, de forma limitada o no, el ya mencionado recurso. Para conseguir acceso a un recurso, el cliente se lo solicita al servidor. El servidor pedirá la aprobación al dueño del recurso. Si el dueño del recurso autoriza la petición, entonces el servidor enviará un *token* de autorización al cliente. El cliente deberá realizar todas sus peticiones utilizando este *token*. El servidor tendrá un mapeo entre *tokens* y privilegios o limitaciones sobre los recursos que provee. Cuando el cliente realice una petición, el servidor comprobará que el *token* recibido tenga acceso al recurso solicitado. Si no lo tiene, denegará el acceso al cliente.

Recurso protegido

Un **recurso protegido** es un recurso almacenado en el servidor para el cual se precisa el permiso explícito del dueño para poder acceder a éste. Es el servidor quien comprobará los permisos de cada recurso, y solicitará la autenticación necesaria.

Un recurso protegido generalmente son datos o funcionalidades específicas de una aplicación. Ejemplos de datos podrían ser los correos electrónicos del usuario en una aplicación *webmail*. Ejemplos de funcionalidades podría ser enviar o recibir correos bajo el nombre del dueño del recurso.

Credenciales y *tokens*

Las credenciales consisten de un identificador único junto con una clave compartida. La especificación de OAuth utiliza tres tipos de credenciales[1]:

- *Client credentials*
- *Temporary credentials*
- *Token credentials*

Las *client credentials* se utilizan para autenticar al cliente de software que quiera utilizar recursos protegidos. Además de autenticar al cliente, esta clase de credenciales permite al servidor llevar un registro de cuáles son los clientes de software que utilizan el servicio, e informar al dueño del recurso acerca de los clientes que buscan acceder a sus recursos protegidos.

Las *temporary credentials*, como su nombre lo indica, son credenciales temporales que se utilizan para identificar el requerimiento de autorización por parte del cliente. Con el objetivo de proveer funcionalidad a distintos clientes, las *temporary credentials* ofrecen una capa adicional de flexibilidad y seguridad.

Finalmente, las *token credentials* se utilizan para identificar el acceso otorgado por el dueño del recurso a sus recursos protegidos. El servidor mantiene un mapeo entre *token credentials* y recursos protegidos habilitados para ese *token*.

A diferencia de la autenticación *Basic* de HTTP, donde la contraseña se envía en cada requerimiento, OAuth utiliza firmas digitales. De esta forma se permite autenticar el requerimiento y verificar que no haya sufrido ningún cambio desde su emisión. La clara ventaja de utilizar firmas digitales en lugar de una

[1] Con el fin de mantener la consistencia se conservan los nombres en inglés

contraseña es que la parte secreta de la credencial nunca será transmitida por el medio, y siempre vivirá en el dispositivo del usuario.

OAuth permite utilizar el paradigma tradicional de las firmas digitales, donde se utiliza criptografía asimétrica para realizar la firma, pero también permite que la firma se haga con la clave compartida de la credencial correspondiente, es decir, utilizando criptografía simétrica. OAauth define estos mecanismos de autenticación como RSA-SHA1 y HMAC-SHA1, respectivamente, y son los que se utilizan cuando el protocolo de aplicación subyacente es inseguro, como HTTP.

Es recomendable utilizar firmas digitales cuando la autenticación se realice por un medio inseguro. Sin embargo, cuando la autenticación del cliente se realice desde un medio seguro como HTTPS, las firmas digitales producen *overhead* innecesario puesto que, en este caso, la capa de transporte provee la seguridad requerida vía SSL. Para estos casos, OAuth implementa el método de autenticación PLAINTEXT, que delega prácticamente todos los requerimientos de seguridad a HTTPS.

Historia y uso

El proyecto de OAuth nació de la comunidad de desarrolladores de OpenID, a fines del 2006, cuando se estaba buscando implementar OpenID en Twitter, pero sin la necesidad de que el usuario deba compartir sus credenciales de acceso con aplicaciones de terceros. Cuando las limitaciones de OpenID en cuanto a la autorización se hicieron evidentes, un grupo de expertos de diversas empresas concluyeron que era necesario definir un nuevo estándar abierto de autenticación que no requiera que el usuario revelara su contraseña, y que sea independiente del sistema de *login* utilizado. Esta nueva iniciativa se llamó **OpenAuth**, inspirados en el nombre de OpenID.

En abril de 2007 la empresa AOL introdujo un nuevo sistema de autenticación denominado OpenAuth que nada tenía que ver con la iniciativa abierta mencionada, por lo que el grupo debió cambiar el nombre de su proyecto. En mayo de 2007, el nombre **OAuth** fue elegido. Finalmente, a fines de 2007 se presentó la especificación de OAuth Core 1.0[31], que luego, con algunos retoques, se convertiría en la RFC 5849[17]. En el año 2012 se publica la especificación de OAuth 2.0[20].

Actualmente versiones modificadas de OAuth son utilizadas por las principales redes sociales de Internet, como Facebook y Twitter.

2.1.3. OpenID Connect y *logins* sociales

La última metodología de autenticación que se analizará será el *login* social. Este tipo de autenticación utiliza información provista por una red social, como Facebook o Google+, para identificar a sus usuarios. Los beneficios de este sistema son similares a los de OpenID. El usuario de estas redes sociales no necesita crear una cuenta nueva para cada nuevo sitio que visita, y además los desarrolladores de los sitios que implementen este tipo de autenticación podrán disponer de datos más acertados sobre sus usuarios[27]. Una de las principales diferencias es, por supuesto, que los beneficios reales de este sistema de autenticación son explotados cuando el proveedor es una red social.

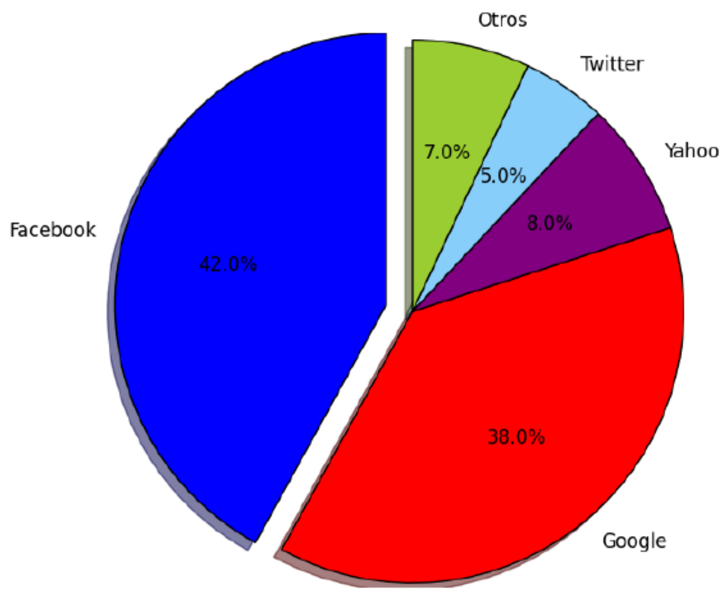


Figura 2.1: Estadísticas de uso de *logins* sociales

La figura 2.1 muestra las estadísticas de uso[2] de *login* sociales durante el primer cuatrimestre del año 2014.

No hay una tecnología estandarizada que defina cómo implementar un *login* social. Por el contrario, cada proveedor lo implementa como quiere, pero en general se tiende a utilizar modificaciones propias de los ya mencionados OAuth y OpenID. Actualmente la entidad detrás de la especificación de OpenID está tratando de que OpenID Connect[34] tome difusión sobre los *logins* sociales.

[2]El análisis estadístico fue realizado por la empresa *janrain*. Más información en: `http://janrain.com/blog/social-login-trends-across-web-q4-2013/`

En términos técnicos, OpenID Connect agrega a OAuth 2 una capa adicional de autenticación, con la capacidad de manejar identidades. Las principales diferencias con OpenID 2.0 residen en que el nuevo estándar utiliza un manejo de mensajes del tipo REST/JSON con el objetivo de que la interfaz sea más clara y limpia que la de su predecesor, que utilizaba XML.

El estándar final de OpenID Connect fue publicado el 26 de febrero de 2014[38]. Varias empresas, incluida Google con su red social Google+[14], están utilizando sistemas de autenticación compatibles con OpenID Connect.

2.2. Problemas actuales en la autenticación

Se han identificado tres defectos en el uso de las tecnologías de autenticación analizadas:

- La centralización de credenciales
- La homogeneidad de credenciales
- Utilizar una entidad de autenticación externa

En lo que resta del capítulo se explicará por qué se considera que cada una de estas características de los actuales sistemas de autenticación con propiedades de *Single Sign-On* pueden llegar a comprometer la privacidad del usuario.

2.2.1. Centralización de credenciales

Cuando se habla de centralización de credenciales se entiende que una única entidad almacena la información de acceso de más de un usuario. Este es el paradigma clásico que se utiliza en sitios web, donde el mismo servidor que provee un servicio almacena la información de sus usuarios junto con sus contraseñas.

En el ámbito de las tecnologías de autorización con propiedades de *Single Sign-On* analizadas, el paradigma recién expuesto difiere en que las credenciales son almacenadas en el servidor del proveedor de autenticación, por ejemplo, un proveedor de OpenID. Sin embargo, se debe notar que las credenciales, ahora dispersas entre varios proveedores de OpenID, si se continúa con el ejemplo, siguen conviviendo con otras credenciales. Aún peor, en este caso un único servidor centraliza las credenciales de acceso de sus usuarios a sitios distintos.

Se considera que centralizar las credenciales en un único servidor es riesgoso, ya que convierte al servicio en un blanco de ataques que intentarán adquirir esta información. Siguiendo buenas prácticas en seguridad informática el riesgo de que las credenciales sean robadas se reduce considerablemente. Sin embargo, de forma constante se descubren nuevas vulnerabilidades que, si no se arreglan en tiempo y forma, provocarán que existan ventanas de tiempo donde el servicio sea vulnerable a ataques externos.

En conclusión, la centralización de credenciales no es un problema en sí mismo. El problema son las consecuencias de que el servidor donde residen se vea comprometido por un atacante y que éste adquiera las credenciales de los usuarios. Si las credenciales están distribuidas, el impacto de un ataque se reduce considerablemente ya que será más difícil para el atacante impersonalizar a los usuarios. Por lo tanto, se considera conveniente que un sistema de autenticación esté diseñado para que las credenciales de sus usuarios no se centralicen en un único servicio.

2.2.2. Homogeneidad de credenciales

Otro problema presente en el paradigma actual de autenticación por Internet es la homogeneidad de credenciales. Este concepto hace referencia a la característica de que un usuario utilice la misma credencial para autenticarse en distintos sitios. El caso más común, por supuesto, es el uso de una misma contraseña para ingresar en distintos sistemas. A pesar de que constantemente se sugiere utilizar distintas contraseñas para cada nueva cuenta creada, este comportamiento es muy común cuando el usuario posee muchas cuentas, y se da naturalmente debido a la dificultad en recordar grandes cantidades de contraseñas.

La situación que convierte a la homogeneidad de credenciales en un problema se da cuando un atacante compromete la base de datos de usuarios de un sitio. Si las credenciales de este sitio consistieran en usuarios y contraseñas, y las contraseñas no estuvieran *hasheadas*, entonces el atacante podría averiguar qué otras cuentas tiene un usuario víctima de la base de datos robada e intentar acceder utilizando la contraseña adquirida. Si las contraseñas estuvieran *hasheadas*, el atacante deberá realizar ataques de *bruteforcing* sobre cada *hash* para adquirir la contraseña en texto plano. Dependiendo de la fortaleza de la contraseña, el atacante podrá o no romper la clave. Aún con las contraseñas *hasheadas*, el atacante estará más cerca de comprometer el resto de las cuentas de una víctima.

En los sistemas *Single Sign-On* este problema se ve solventado debido a que se delega la tarea de autenticación a un proveedor dedicado a tal fin. Es de suponer

que un proveedor de *Single Sign-On* seguirá buenas prácticas al momento de implementar su sistema, y por lo tanto las contraseñas de sus usuarios estarán a salvo almacenadas de forma *hasheada*. De esta forma, si el usuario sigue buenas prácticas, su contraseña estará almacenada de forma segura únicamente en la base de datos del proveedor de *Single Sign-On*.

Sin embargo, se considera que la solución al problema planteado es parcial ya que, en el caso de los sistemas *Single Sign-On*, se utiliza la misma credencial para acceder a distintos sitios. En consecuencia, el compromiso de esta clave implicaría, una vez más, el compromiso total de las cuentas asociadas. Esta situación es indeseable y, por lo tanto, al momento de diseñar un sistema de autenticación se debe considerar evitar la homogeneidad de las credenciales.

2.2.3. Entidad de autenticación externa

El último problema a analizar en los sistemas de autenticación actuales en el ámbito de Internet consiste en la confianza que deposita un usuario a una entidad de autenticación externa. En los sistemas de *Single Sign-On* como OpenID, cada vez que se desea ingresar a un nuevo sitio, éste debe solicitar permiso al proveedor para conocer la identidad del usuario. Si el usuario está de acuerdo, entonces el proveedor autorizará el requerimiento del sitio y así éste conocerá la identidad requerida.

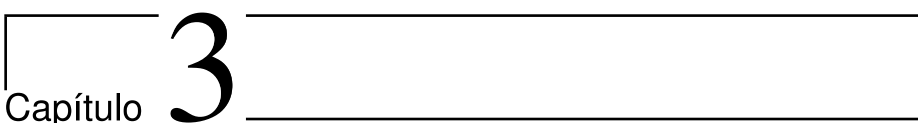
El problema que ocurre bajo este esquema se da producto de la comunicación que debe existir entre el servidor que requiere autenticación y el proveedor de *Single Sign-On*. Cuando ocurre este contacto, inevitablemente el proveedor sabrá en qué momento y a qué sitios acceden sus usuarios, provocando así una importante pérdida de privacidad.

Se puede objetar que en los sistemas de autenticación con estándares abiertos, como OpenID, un usuario podría configurar su propio servidor para administrar su entidad de autenticación. Pero, si bien esto es cierto, se debe tener en cuenta que para llevar a cabo tal acción se requieren conocimientos de administración de servidores que el usuario medio de Internet no dispone. Por ello, aún si OpenID permite federalizar la autenticación, la verdadera independencia sobre las entidades externas solo está al alcance de los usuarios con conocimientos informáticos.

Actualmente la situación de este problema está empeorando ya que, como se mencionó en la sección anterior, cada vez son más los sitios que dejan de implementar OpenID para dar lugar a los *login* sociales. En consecuencia, limitan a los usuarios en sus opciones para realizar *Single Sign-On* a un reducido conjunto

de redes sociales, sin siquiera darles la posibilidad de configurar su propio servidor como era el caso de OpenID. La situación tal vez se revierta con el advenimiento de OpenID Connect, pero en el mejor de los casos se volverá a la situación descripta en el párrafo anterior.

Así pues, es deseable que al momento de diseñar un sistema de autenticación, sus usuarios no deban depender de entidades externas que fácilmente puedan registrar la información sobre los sitios que visitan.



Solución propuesta

Este capítulo presentará las soluciones propuestas a las problemáticas planteadas en el capítulo anterior. Se llegará a la conclusión de que se precisa cierta infraestructura para implementar el sistema de autenticación propuesto, por lo cual se describirán sus componentes y se explicará la arquitectura de una posible implementación.

3.1. Solución a la problemática

La solución propuesta a los problemas expresados en el capítulo anterior se enfoca en atacar las desventajas de la centralización de credenciales, de la homogeneidad de credenciales, y de la delegación de la autenticación a una entidad externa. Para solucionar el primer aspecto, inevitablemente se deberán alojar partes de las credenciales de autenticación en el sistema operativo del cliente. Para lograr el segundo cometido, se deberá disponer de un método que permita generar credenciales distintas, preferentemente para cada nuevo sitio en el que el usuario precise una cuenta, sin que ello provoque la desventaja de recordar algo nuevo. Finalmente, con el fin de eliminar la necesidad de una entidad de autenticación externa, el cliente deberá implementar cierta infraestructura con el fin de permitir que el servidor que requiera autenticación lo identifique.

3.1.1. Centralización de credenciales

Como ya se ha mencionado, la centralización de credenciales es un factor importante a tener en cuenta en el diseño de un sistema de autenticación. En el ámbito de Internet, centralizar las credenciales del lado de quien provee el servicio tiene como ventaja el poder autenticarse desde cualquier dispositivo sin necesidad de pre-configurarlo. Sin embargo, esta bondad pronto se convierte en vulnerabilidad si el servidor se ve comprometido, puesto que las credenciales de todos sus usuarios, o sus respectivos *hash*, serán expuestos al atacante. Se debe, pues, buscar un método para que las credenciales de los usuarios de un servicio no residan en un único punto, puesto que de lo contrario éste será un constante blanco de ataques.

Para este fin, se propone utilizar criptografía asimétrica. Este tipo de criptografía consiste en la generación de un juego de claves con una característica particular: la información cifrada con una de las claves solo puede ser descifrada con la otra.

Criptografía asimétrica

La criptografía asimétrica tiene sus orígenes en la década del 70. En el año 1976, W. Diffie y M. Hellman publicaron el primer sistema criptográfico de claves asimétricas [6]. Sin embargo, en el año 1997, se reveló que en 1973 James H. Ellis, Clifford Cocks, y Malcolm Williamson, trabajando para la GCHQ (*Government Communications Headquarters*) del Reino Unido, habían desarrollado un sistema criptográfico de similares características[12].

El funcionamiento de la criptografía asimétrica requiere dos claves distintas, una secreta, o *privada*, y la otra *pública*. Aunque estas claves sean distintas, están matemáticamente conectadas. El término *asimétrico* proviene del uso de distintas claves para realizar funciones opuestas. Sin embargo, cabe destacar que tanto la clave pública como la privada puede cumplir la función de su contraparte. La designación de una de las claves como pública y otra como privada es independiente de las propiedades matemáticas de los números subyacentes.

La fortaleza de los algoritmos de criptografía asimétrica reside en la dificultad de resolver determinados problemas matemáticos para los cuales se desconocen soluciones eficientes, tales como la factorización de enteros, la resolución de logaritmos discretos[7], o el uso de curvas elípticas[29]. Esto implica que es computacionalmente sencillo generar un juego de claves asimétricas, pero, por el contrario, computacionalmente difícil determinar una clave a partir de la otra.

La utilidad de la criptografía asimétrica reside en la capacidad de transmitir información cifrada sin la necesidad de compartir una clave secreta con antelación. Esto se logra por la naturaleza misma de este tipo de criptografía, ya que, dado un juego de claves, la designada como pública se puede difundir por canales inseguros sin por ello reducir la efectividad del sistema de cifrado. De esta forma, todo mensaje cifrado con la clave pública podrá ser descifrado únicamente con la clave privada, lo cual proporciona la confidencialidad requerida.

Firma digital

Algunos algoritmos de criptografía asimétrica proveen, además del cifrado básico, la capacidad de **autenticar** al emisor de un mensaje mediante una firma digital. Un mensaje firmado digitalmente permite al receptor adquirir la certeza de que dicho mensaje fue redactado por el dueño de la clave privada, y por lo tanto el emisor no podrá negar el haberlo enviado[1]. Esta facilidad se conoce como **no repudio**. Además, una firma digital proporciona la certeza de que el mensaje original no ha sido modificado. A esta característica se la conoce como **integridad**.

En términos técnicos, una firma digital no es más que un *hash* del mensaje calculado en el momento previo a su envío. El emisor del mensaje cifra dicho *hash* con su clave privada, de forma que solo pueda ser descifrado utilizando la correspondiente clave pública. Finalmente, el emisor envía el mensaje acompañado del *hash* cifrado. Por otro lado, para que el receptor pueda comprobar la **autenticidad** e **integridad** del mensaje, deberá calcular su *hash* (haciendo uso, por supuesto, del mismo algoritmo que utilizó el emisor), y además deberá obtener el *hash* enviado por el emisor, descifrando la firma recibida utilizando la correspondiente clave pública. Si ambos *hash* con iguales, entonces el receptor habrá verificado la validez del mensaje.

Aplicación en el sistema de autenticación

La criptografía asimétrica proporciona una pieza fundamental en el funcionamiento del sistema de autenticación propuesto en el presente trabajo. Este tipo de criptografía permite que las credenciales asociadas a un usuario no estén centralizadas en el dispositivo proveedor del servicio. Por el contrario, se propone que las cuentas de usuario estén asociadas a claves públicas en lugar de contraseñas, y que los propios usuarios almacenen sus claves privadas. De esta forma, si el servicio se ve comprometido, las credenciales de autenticación

[1] Podría negarlo en caso que su clave privada hubiera sido comprometida

de los usuarios no se verán afectadas, pues las correspondientes claves privadas residirán en los dispositivos de los clientes.

Para lograr autenticar usuarios de esta forma, se debe buscar un método que permita verificar que quien quiere autenticarse es el poseedor de la clave privada asociada a la clave pública del usuario con el cual se está queriendo autenticar. El método que se eligió para este sistema de autenticación consiste simplemente en generar un *challenge*, es decir, una suerte de contraseña, y cifrarlo con la clave pública del usuario que pretenda autenticarse. De esta forma, mediante las propiedades de la criptografía asimétrica, se puede asegurar que solo utilizando la correspondiente clave privada se podrá descifrar el *challenge*. El *challenge* cifrado será enviado por un medio pre-acordado al usuario, para que luego éste lo descifre y envíe su respuesta utilizando el mismo medio. El servidor, luego de recibir la respuesta del usuario, deberá comprobar si coincide con el *challenge* generado anteriormente, y que la respuesta haya llegado antes de un tiempo pre-establecido. Si estas condiciones se cumplen, entonces el servidor habrá confirmado que quien quiere autenticarse con una cuenta de usuario, es el poseedor de una de las claves privadas asociadas a ésta.

En términos técnicos, el funcionamiento del sistema de autenticación propuesto es muy similar al utilizado en el protocolo SSH[10]. En la autenticación RSA de SSH, cuando un cliente desea conectarse con un servidor, este último le comunica su *fingerprint*, una suerte de resumen de su clave pública. Si es la primera vez que el cliente se conecta al servidor, entonces deberá verificar manualmente que el *fingerprint* sea válido. Si lo es, entonces guardará la asociación entre el nombre de *host* o IP del servidor, y el *fingerprint* recibido, con el objetivo de que las futuras verificaciones se hagan de forma automática. Luego de verificada la autenticidad del servidor, el cliente comunicará su nombre de usuario junto con su clave pública, o su correspondiente *fingerprint*. El resto del proceso no difiere del propuesto.

3.1.2. Homogeneidad de credenciales

Como se ha mencionado anteriormente, los usuarios de numerosas cuentas tienden a repetir sus credenciales, generalmente contraseñas, o patrones para generar contraseñas, en sitios que nada que ver tienen entre sí. Este comportamiento se da naturalmente debido a la dificultad de recordar contraseñas complejas, difíciles de vulnerar utilizando una metodología de fuerza bruta. Como se explicó en la sección anterior, si un servicio se ve comprometido, y un atacante logra robar una base de datos con cuentas de usuario y contraseñas, ya sea en forma de *hash* o en texto plano, entonces dicho atacante podría elegir una

víctima y averiguar qué otros servicios utiliza. Si las contraseñas adquiridas están almacenadas en texto plano, entonces el atacante podría inmediatamente probar suerte en otros servicios que utilice el usuario, como, por ejemplo, su cuenta de correo electrónico. En caso de que el atacante solo hubiera conseguido los *hash* de las contraseñas, se añadiría la dificultad de deducir el algoritmo de *hash* utilizado y lanzarles ataques de fuerza bruta. Si la contraseña de la víctima fuera buena y de una longitud aceptable (donde buena significa que tenga una combinación de letras mayúsculas, minúsculas, símbolos y números, y longitud aceptable significa que tenga ocho o más caracteres), entonces probablemente las intenciones del atacante se verían frustradas, pero en caso de no ser así, entonces la contraseña se podría romper con facilidad y podría continuar el procedimiento ya mencionado.

Trivialmente, con la solución propuesta este problema se evita completamente, pues del lado del servidor solo se almacena información pública, insuficiente para que quien ataca pueda comprometer otras cuentas de una víctima. Sin embargo, lo que un atacante sí puede lograr es identificar las diversas cuentas de un mismo usuario a partir de su clave pública. Es decir, si el usuario optara por tener varias cuentas, algunas con un nombre de usuario que identifique su persona, y otras con un seudónimo, y a todas asociara la misma clave pública, entonces su anonimato se vería fácilmente comprometido por todo atacante que tuviera acceso a por lo menos dos bases de datos, una con su usuario real y otra con su seudónimo. Por tanto, para evitar esta debilidad, la solución a implementar debe tener como requisito la capacidad de generar nuevas credenciales para cada cuenta, y evitar que por ello el usuario deba memorizar nueva información. Esto se logra simplemente asociando un juego de claves distinto a cada nueva cuenta que cree el usuario. Por las propiedades de la criptografía asimétrica, el usuario no deberá recordar nueva información, como una contraseña, y cada una de sus cuentas será completamente independiente de las otras, lo que resultará en el anonimato requerido.

3.1.3. Entidad de autenticación externa

Hoy día, el problema ya expuesto de memorizar una gran cantidad de contraseñas se tiende a solucionar mediante la delegación de la autenticación a una entidad externa. De esta forma, el usuario mantiene una cuenta en dicha entidad y asocia el resto de sus cuentas con ella. Como se mencionó anteriormente, esta solución es cuestionable desde el punto de vista de la privacidad del usuario, pues inevitablemente la entidad de autenticación externa conocerá todos los sitios que éste visita, la hora a la que accede a ellos, entre otros datos.

Por tanto, la solución propuesta tiene como requisito la no delegación de la autenticación a una entidad externa. Por el contrario, cada usuario será responsable de sus propias credenciales de acceso. Esto requiere una aplicación que haga las veces de infraestructura para almacenar las claves y se comunique con los servidores que precisen autenticar a sus usuarios.

3.2. Arquitectura de la infraestructura

La problemática planteada en la sección anterior se ve solucionada mediante el diseño y uso de una aplicación que permita al usuario almacenar sus claves privadas y comunicarse con los servidores que requieren autenticarse. Como parte del presente trabajo de grado se ha diseñado e implementado esta aplicación. En lo que resta del capítulo se explicarán las decisiones de diseño tomadas.

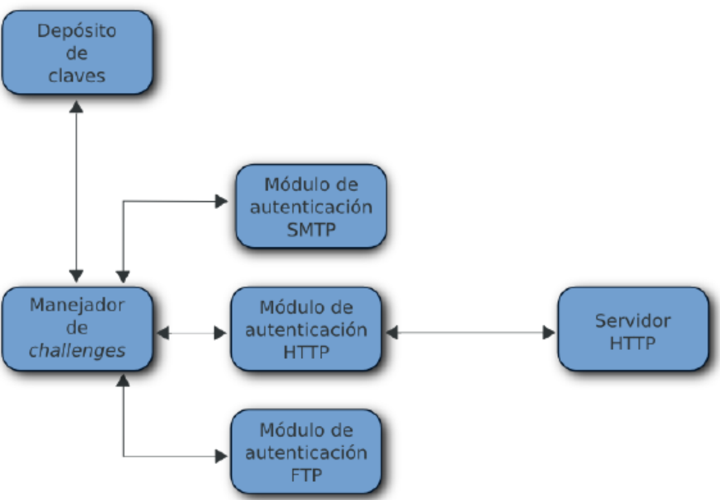


Figura 3.1: Intraestructura

La figura 3.1 muestra una visión general de las partes que componen el sistema. A continuación se explicará detalladamente cada uno de estos componentes.

3.2.1. Depósito de claves

Se debe disponer de algún método para almacenar las claves privadas. Para ello, se precisa que el sistema de autenticación haga uso de un nuevo componente: el **depósito de claves**. Este depósito de claves deberá tener la funcionalidad de generar y almacenar credenciales de autenticación, en particular, juegos de

claves asimétricas. Aunque no es un requisito, este depósito de claves podrá guardar las claves privadas de forma cifrada, en este caso utilizando algún algoritmo de criptografía simétrica, como AES o DES3. De esta forma se obtiene una funcionalidad similar a la de los sistemas *Single Sign-on*, donde el usuario solo escribe una única contraseña, en este caso para descifrar el depósito de claves privadas, para luego poder autenticarse libremente con el resto de sus cuentas.

El funcionamiento general del depósito de claves se describe en el diagrama de la figura 3.2.

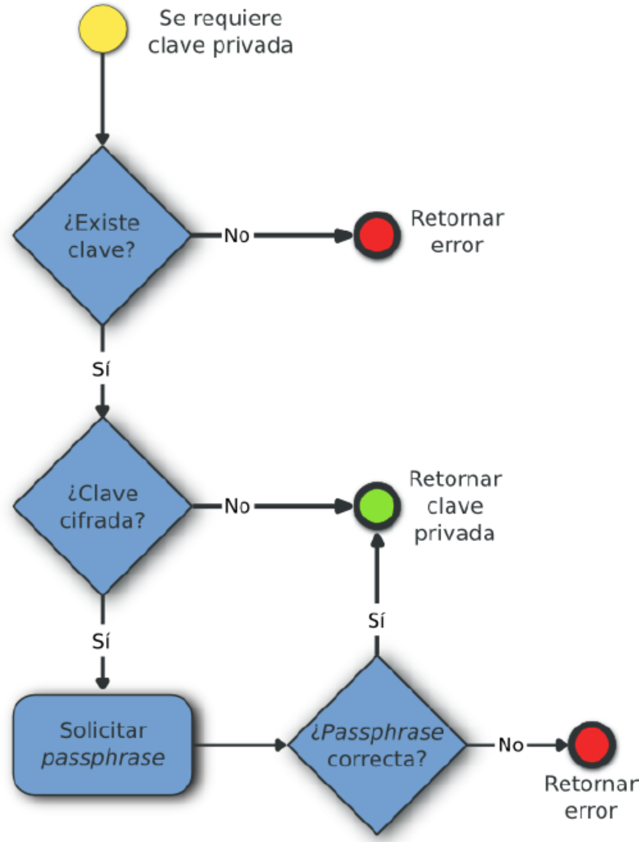


Figura 3.2: Depósito de claves

3.2.2. Manejador de *challenges*

Todo servidor que requiera autenticar a un usuario deberá enviarle un *challenge*, que no es más que una contraseña generada aleatoriamente cifrada con una de las claves públicas del usuario. Cuando un cliente recibe un *challenge*, deberá

solucionarlo, es decir, descifrarlo utilizando la clave privada adecuada, y enviar la respuesta al servidor. La funcionalidad de enviar y recibir la información corresponde a los módulos de autenticación, y será explicada en la siguiente sección. Cuando el módulo de autenticación recibe un *challenge*, éste no se comunica directamente con el depósito de claves, puesto que este comportamiento sería común a todos los módulos de autenticación y la repetición de código sería inevitable. Por ello, entonces, surge la necesidad de añadir un nuevo componente a la arquitectura de la aplicación, el **manejador de *challenges***.

Este nuevo componente se encarga de recibir los *challenges* adquiridos por los módulos de autenticación, solicitar la clave privada adecuada al depósito de claves y, en caso de ser posible, descifrar la respuesta. Luego de descifrada, la respuesta es transmitida hacia el módulo de autenticación correspondiente. La clave privada deberá ser inmediatamente descartada, puesto que su retención en forma descifrada vulnera las seguridades provistas por el depósito de claves.

El depósito de claves y el manejador de *challenges*, a pesar de tener una funcionalidad claramente separada, podrían ser implementados como un único componente y de esta forma evitar el *overhead* que produce transmitir los requerimientos de uno a otro. También, de esta forma, se evitarían posibles filtraciones de claves privadas producto de una implementación deficiente del canal de comunicación entre ambos componentes. Por otro lado, disponer de estos módulos de forma separada permite que puedan ser reemplazados y actualizados de forma independiente. Así pues, las dos opciones son correctas, y la decisión final quedará a cargo de quien implemente el sistema.

El funcionamiento general del manejador de *challenges* se describe en el diagrama de la figura 3.3.

3.2.3. Módulos de autenticación

El sistema de autenticación propuesto es independiente del protocolo subyacente que se utilizará para realizar la autenticación. Por lo tanto, todo componente del sistema fue diseñado de forma genérica, evitando de esta forma la necesidad de atarse a un protocolo. Así, el sistema dispondrá de componentes independientes que podrán ser reutilizados en todos los protocolos donde se implemente este tipo de autenticación. Sin embargo, al momento de realizar la autenticación, es necesario que el sistema utilice el protocolo subyacente, como HTTP, FTP o SMTP, entre otros, lo que provoca que se diluya la generalidad buscada.

Naturalmente se desprende un nuevo componente del sistema: el **módulo de**

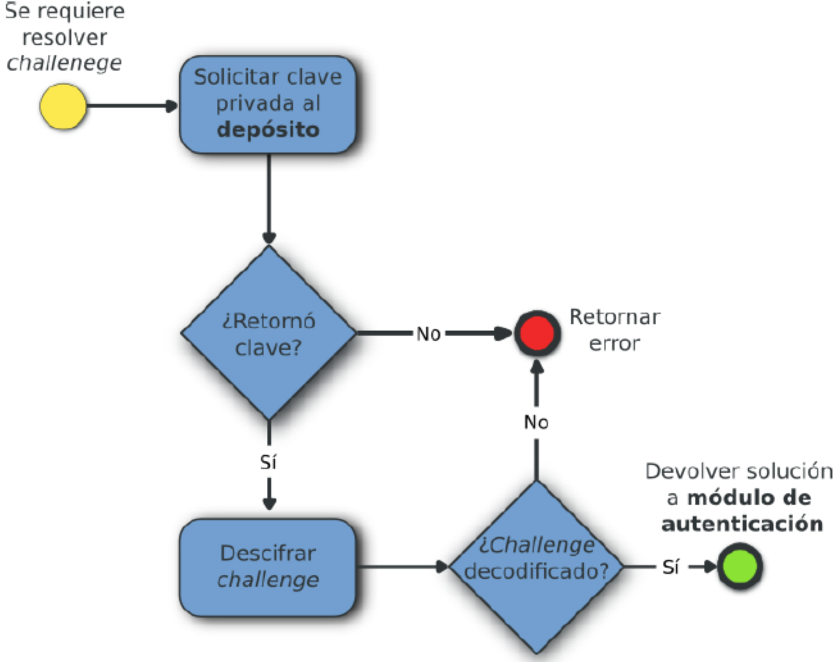


Figura 3.3: Manejador de *challenges*

autenticación. Un módulo de autenticación se encargará de encapsular todo el comportamiento dependiente del protocolo subyacente. Hará de intermediario entre el servidor y el manejador de *challenges*. Recibirá los *challenges* enviados por el servidor y enviará las soluciones correspondientes.

El canal a utilizar para enviar y recibir datos se implementa enteramente en el módulo de autenticación. Esto significa que, por ejemplo, para HTTP se puede establecer un canal entre el servidor y el módulo de autenticación de HTTP utilizando *websockets*[8] o AJAX[11], para SMTP se podría implementar una extensión de la autenticación, entre otros. Sin embargo, independientemente del módulo de autenticación utilizado, se debe utilizar un lenguaje común que realice el intercambio de *challenges* y soluciones. De esta forma surge la necesidad de otro componente, el **protocolo** de comunicación.

3.2.4. Protocolo

Luego de establecer el canal de comunicación, naturalmente surge la necesidad de idear un idioma común entre el servidor y el módulo de autenticación. Este idioma común no debe acoplarse al protocolo subyacente del canal de comunicación, por lo que se diseñó de forma genérica con el objetivo de que sea

reutilizable por todos los módulos de autenticación.

A continuación se describe el procedimiento general del protocolo:

1. El **cliente** pretende autenticarse y se lo informa al servidor.
2. El **servidor** se comunica con el módulo de autenticación del cliente y le indica que desea iniciar el procedimiento de autenticación.
3. El **cliente** identifica al servidor y consulta su base de datos local para buscar la cuenta asociada a ese servicio. Luego, responde al servidor indicando su nombre de usuario y *fingerprint* de la clave pública asociada a la cuenta.
4. El **servidor** recibe la información enviada por el cliente y comprueba que los datos sean correctos. Es decir, comprueba que el nombre de usuario exista, y que el *fingerprint* recibido se corresponda con el de alguna de las claves públicas asociadas a la cuenta.
5. Si la información es correcta, el **servidor** genera un *challenge* y lo encripta utilizando la clave pública asociada al usuario. El *challenge* puede consistir de letras mayúsculas, minúsculas, números y caracteres especiales. El servidor asigna un *timeout* al *challenge*.
6. El **servidor** envía el *challenge* cifrado al cliente.
7. El **cliente** recibe el *challenge* y lo descifra utilizando la correspondiente clave privada. Finalmente, envía la solución al servidor.
8. El **servidor** recibe la solución, comprueba que el *timeout* asociado no haya vencido, y si la solución es correcta entonces autentica al cliente.

3.2.5. Servidor

El servicio que desee implementar el sistema de autenticación propuesto deberá establecer la comunicación con alguno de los módulos de autenticación del usuario. Por tal motivo, debe haber un preacuerdo sobre la metodología a utilizar para establecer un canal de comunicación. El proceso de autenticación comienza cuando el usuario informa al servidor que quiere autenticarse. El servidor, siguiendo el protocolo propuesto, se identificará a sí mismo para luego recibir por parte del cliente un nombre de usuario junto con uno de sus *fingerprints*. Se deberá validar, entonces, que el nombre de usuario exista y que esté relacionado con el *fingerprint* recibido. En caso afirmativo, el servidor generará de forma aleatoria un *challenge*, y lo asociará a la sesión donde se esté

manteniendo la comunicación con el cliente. Además, agregará un tiempo de expiración relativamente corto para mitigar ataques de fuerza bruta.

Cuando el módulo de autenticación del cliente envíe la respuesta del *challenge*, el servidor deberá comprobar que la solución recibida sea correcta, y que el tiempo de expiración no haya caducado. El tiempo de expiración puede ser corto, no más de una decena de segundos. Si se cumplen ambas condiciones, entonces se afirma que la autenticación se ha realizado de forma correcta. De lo contrario, si la solución es incorrecta, entonces el servidor deberá interrumpir la comunicación y caducar la sesión junto con su *challenge*. Esta medida se toma debido a que el proceso descripto es automático y por lo tanto no admite errores. Si un error de este tipo es detectado, entonces claramente se está realizando un ataque hacia el sistema.

Autenticación del servidor

Se habrá notado, tal vez, que en ningún momento se autentica al servidor. Es decir, cuando el usuario le indica al servidor que quiere autenticarse, éste responde informando quién es para que luego el cliente envíe su usuario y *fingerprint* de alguna de sus claves públicas. Este comportamiento resulta en una excesiva facilidad para realizar ataques de *spoofing* o *man in the middle*, pero se ve rápidamente mitigado si el servidor envía su identidad junto con un *token* generado por el usuario y firmado digitalmente por el servidor.

La solución al problema planteado implica que el servidor también deba manejar un juego de claves asimétricas, lo que agrega la complejidad inherente de mantener de forma segura una clave privada del lado del servidor. Por ello, siempre que sea posible se autenticará al servidor utilizando el mecanismo subyacente de la conexión. Es decir, en el caso del módulo HTTP, para que la conexión evite ataques de *spoofing* se deberá utilizar la infraestructura de HTTPS. De forma genérica, se podría usar SSL para todos los protocolos que lo soporten, habiendo previamente verificado el correspondiente certificado.

Cuando el protocolo subyacente no provea autenticación será necesario autenticar el servidor de otra forma. Para ello se podría utilizar una extensión al protocolo propuesto que, como se mencionó anteriormente, autentique al servidor mediante un nuevo juego de claves asimétricas instaladas del lado del servidor. El procedimiento consistiría en que el cliente genere de forma aleatoria un *token* y se lo envíe al servidor, para que éste luego de recibirlo lo devuelva firmado digitalmente. De esta forma, como el usuario dispone previamente de la clave pública del servidor, podrá comprobar que la firma digital sobre el *token*

recibido proviene realmente del servidor, y de esta forma se podrá continuar normalmente con el protocolo de autenticación. Este mecanismo es análogo a las *hosts keys*[43] que provee la arquitectura de SSH.

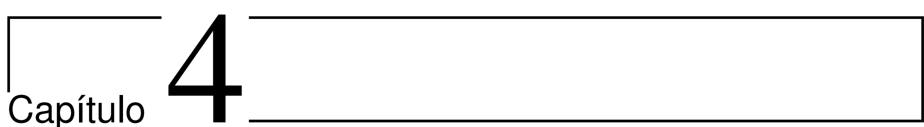
En el peor de los casos, si de alguna forma un ataque de *spoofing* se ha concretado, la información confidencial del usuario no se verá expuesta. La información revelada consistirá en el usuario y el *fingerprint* de una de las claves públicas asociadas al sistema. El mayor recurso al que el atacante podrá apelar ante estas circunstancias será un ataque de *phishing*.

3.3. Autenticación móvil

Así como el uso de contraseñas requiere de la buena memoria del usuario para recordarlas, el uso de criptografía asimétrica requiere de un dispositivo de memoria secundaria donde alojar los juegos de claves. Inevitablemente este funcionamiento limitará al usuario a autenticarse únicamente desde los dispositivos donde se dispongan las claves privadas asociada a la cuenta. Esto, que en principio puede verse como una gran desventaja, provee una capa adicional de seguridad puesto que se asegura que solo se ingresará desde dispositivos habilitados. Además, el surgimiento masivo de dispositivos móviles, como computadoras portátiles y teléfonos celulares, genera la tendencia de ingresar a las cuentas personales desde un mismo dispositivo.

Sin embargo, para los casos donde sea necesario autenticarse desde un dispositivo que no contenga una clave asociada, se ha ideado un mecanismo que soluciona esta carencia. Se toma como pre-condición que el usuario dispone de un teléfono moderno con cámara digital (hace unos años atrás esto podría haber sido una limitación importante, pero no ocurre lo mismo hoy día). Entonces, cuando el usuario indique que quiere autenticarse, el servidor ofrecerá la posibilidad de hacerlo de forma *móvil*. Para ello, el usuario previamente habrá designado una de sus claves públicas para que sea la elegida al utilizar el teléfono para autenticarse. Cuando el usuario selecciona que quiere autenticarse de forma móvil, el servidor buscará la clave pública designada para tal fin y generará un *challenge* para luego cifrarlo con la clave pública y codificarlo como un código QR[24]. Luego, el usuario utilizará una aplicación móvil desde su teléfono que utilice la cámara para obtener el código QR, lo decodifique, y finalmente lo descifre utilizando la correspondiente clave privada previamente cargada. Finalmente, el usuario deberá ingresar manualmente en el método de entrada elegido por el servidor la solución al *challenge* descifrado, análogamente a una contraseña común de un solo uso.

La aplicación móvil podrá obtener las claves privadas del usuario importándolas de la misma forma que recibe los *challenges*, utilizando códigos QR. Para ello, el depósito de claves deberá implementar la funcionalidad que permita exportar las claves privadas utilizando este tipo de codificación. La aplicación móvil también podrá optar por generar nuevos juegos de claves, aunque se debe tener en cuenta en estos casos que la asociación de las claves públicas a las cuentas se deberá hacer de forma manual.



Implementación y resultados

La realización y puesta en marcha de la solución descripta en el capítulo anterior requirió del desarrollo de tres componentes bien definidos. Cada componente fue desarrollado utilizando un lenguaje de programación distinto, apto para la solución al problema y la plataforma donde se ejecuta.

- **Servicio local**: las principales funciones de este servicio son actuar como depósito de claves y comunicarse con los servidores que requieran autenticar a sus clientes.

  El almacén de claves es una aplicación web local, por lo cual es independiente a la interfaz gráfica que esté utilizando el usuario. Esta aplicación permite al usuario generar juegos de claves asimétricas, y asociarles cuentas en distintos sitios. También permite exportar los juegos de claves a códigos QR, con el objetivo de que el usuario pueda importarlos en dispositivos móviles.

  Por otro lado, el servicio local posee un servidor *websocket* que escucha los requerimientos de los servidores que requieran autenticarse. Esta parte de la aplicación es la encargada de actuar como intermediario entre el depósito de claves y estos servidores.

  El desarrollo de este servicio se realizó utilizando el lenguaje de programación *Haskell*.

- **Servidor de ejemplo**: se desarrolló un sencillo sitio con una sección restringida a usuarios autenticados, una opción para registrarse en el sistema y, por supuesto, la opción de autenticarse utilizando el sistema desarrollado.

El desarrollo de esta aplicación web se realizó utilizando los lenguajes de programación *PHP* y *Javascript*.

- **Aplicación para dispositivos móviles**: se desarrolló una aplicación que permite importar juegos de claves. La importación se realiza mediante códigos QR leídos utilizando la cámara del dispositivo. La aplicación, además, permite al usuario leer *challenges* encriptados, utilizando la misma metodología de códigos QR, y desencriptarlos, siempre que sea posible, utilizando las claves privadas asociadas.

  La aplicación está desarrollada para dispositivos móviles que utilicen el sistema operativo *Android* y, por lo tanto, se programó utilizando el lenguaje de programación *Java*.

Los tres desarrollos están liberados bajo la *GNU General Public License*.

4.1. Servicio local

Como se ha mencionado anteriormente, el uso del mecanismo de autenticación propuesto requiere de una herramienta que se ejecute en el dispositivo del cliente. Esta herramienta cumple el rol de depósito de claves y manejador de *challenges*. En otras palabras, deberá encargarse de almacenar las claves privadas del usuario y de recibir y solucionar los *challenges* enviados por los módulos de autenticación.

4.1.1. *Frontend* de la aplicación

Si bien existen diversos tipos de interfaces para programar una aplicación con estas características, se decidió utilizar una arquitectura web con el objetivo de que su *look and feel* sea el mismo en todas las plataformas donde se ejecute, y su portabilidad no requiera esfuerzo alguno.

La aplicación está compuesta por tres módulos cuya funcionalidad está claramente separada:

- *Dashboard*
- Claves
- Cuentas

Por defecto la aplicación web escucha peticiones en el puerto 3000. Por lo tanto, cuando el usuario intente acceder a la aplicación deberá escribir en la

barra de direcciones de su navegador web la URL `http://localhost:3000`. Los módulos mencionados podrán ser accedidos mediante enlaces a partir de esta URL.

Dashboard

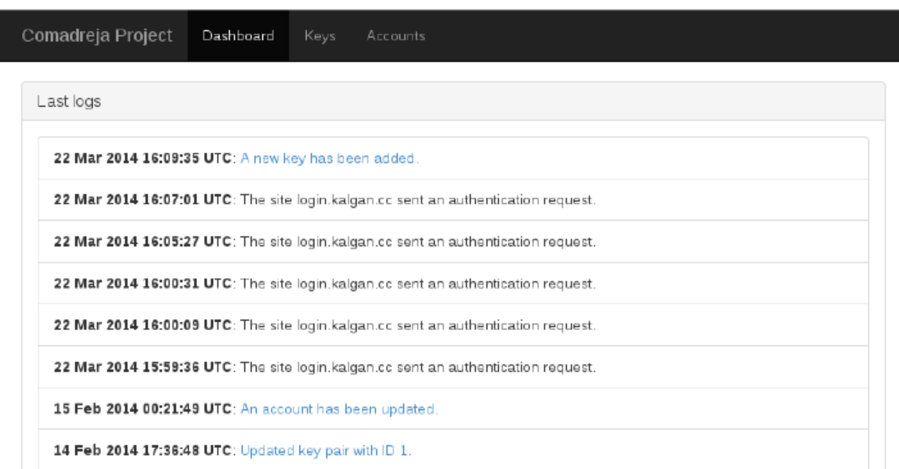


Figura 4.1: Captura de pantalla del *Dashboard*

La figura 4.1 muestra la pantalla de inicio de la aplicación web. La pantalla de inicio muestra un registro de las últimas actividades registradas por la aplicación. Estas actividades comprenden la inserción, actualización y borrado de juegos de claves asimétricas y cuentas, y los requerimientos de autenticación por parte de sitios externos.

Siempre que sea posible, la aplicación generará enlaces hacia los objetos anteriormente mencionados. Por ejemplo, si se agrega o actualiza un juego de claves asimétricas, el registro que aparece en pantalla contendrá un enlace hacia el módulo **Claves**, donde se podrá obtener más información acerca de este elemento. Si en algún momento el juego de claves es borrado, los registros se actualizarán y eliminarán el enlace, y de está forma la aplicación se mantendrá libre de enlaces rotos.

La aplicación registra, además, el horario en el que sucedieron los eventos. En la figura 4.1 se puede ver, por ejemplo, que el día 22 de marzo de 2014 se agregó un nuevo juego de claves asimétricas.

Desde la barra de navegación superior se puede acceder al resto de los módulos de la aplicación: **Claves** y **Cuentas**.

Claves

Keys loaded in the system

Public key	Comment	Actions
DED0 9BDF C33D CE6A B119	:-|	
95AC F086 25FF 63CE B3EB	:-(	
2030 8417 D027 21B3 B655	:-)	
3A25 8575 0978 DB46 E45F	:-/	
0D1F 3BA7 7F79 1778 6537	:-S	

Figura 4.2: Captura de pantalla del módulo **Claves**

La figura 4.2 muestra la pantalla inicial del módulo **Claves**. Desde este apartado se puede obtener una visión global de todos los juegos de claves asimétricas que la aplicación conoce. Se muestra un listado con los *fingerprints* de todas las claves públicas, y para cada uno se provee un enlace donde se puede obtener más información de la clave. Además, sobre cada clave se muestra un pequeño comentario opcional que permite identificar a una clave mediante alguna frase más nemotécnica que su *fingerprint*.

Desde este apartado, además, se podrá editar o eliminar cualquier juego de claves presionando los botones verdes y rojos respectivamente. El botón naranja permite generar un código QR del juego de claves seleccionado, dejándolo listo para ser exportado a un dispositivo móvil con cámara. La figura 4.3 muestra una captura de pantalla con un código QR generado a partir de esta acción.

La figura 4.4 muestra una captura de pantalla con la información específica de un juego de claves asimétricas. Se permiten ver los valores de las claves públicas y privadas en base 10, y el comentario descriptivo elegido por el usuario. Aunque no se muestra en la figura, desde este apartado también se pueden observar los nombres de las cuentas asociadas a esa clave pública, y para cada una de ellas se provee un enlace que apunta a una página con más información.

La figura 4.5 muestra una captura de pantalla del apartado para crear un nuevo juego de claves asimétricas, siempre dentro del módulo **Claves**. Este apartado permite agregar un juego de claves a partir de uno existente ingresando manualmente cada uno de los campos, o bien auto-generar uno nuevo. En este último caso, el único campo que el usuario podrá modificar va a ser el del comentario descriptivo, puesto que modificar el resto carece de utilidad. A continuación se

Figura 4.3: Captura de pantalla de un juego de claves exportado

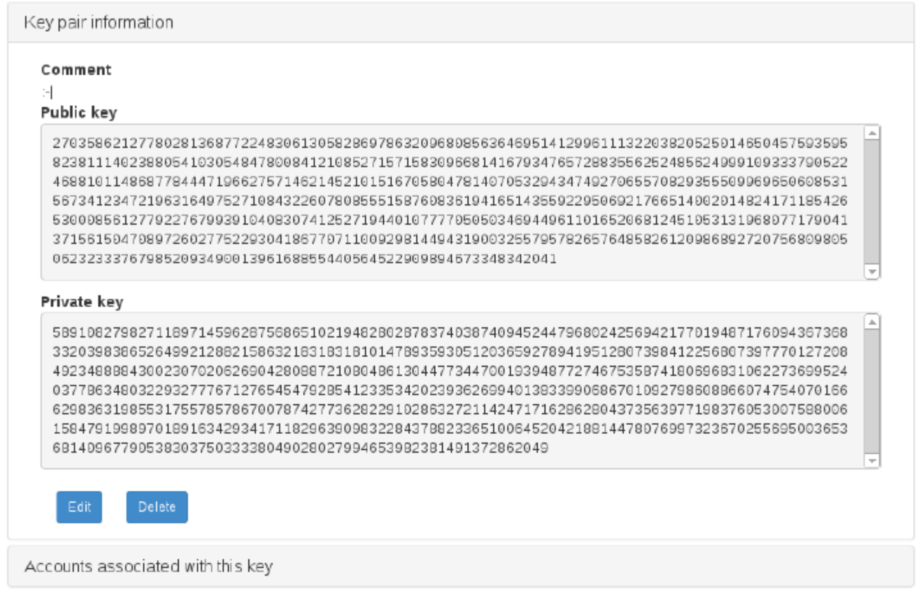


Figura 4.4: Capturas de pantalla con información acerca de un juego de claves

describe el resto de los campos:

- `Key type`: Tipo de clave. Como se explicará posteriormente, es posible crear claves con distintas características y algoritmos. Este campo permite elegir entre las posibilidades implementadas.
- `Key size`: Tamaño en bits de la clave. Cuando se auto-genera un nuevo juego, por defecto se utiliza 2048.
- `Public N`: El módulo o, en otras palabras, la clave pública.
- `Public E`: El exponente público.
- `Private D`: La clave privada.
- `Fingerprint`: El *fingerprint* se calcula automáticamente tomando los últimos veinte números hexadecimales de la clave pública y separándolos cada cuatro unidades.

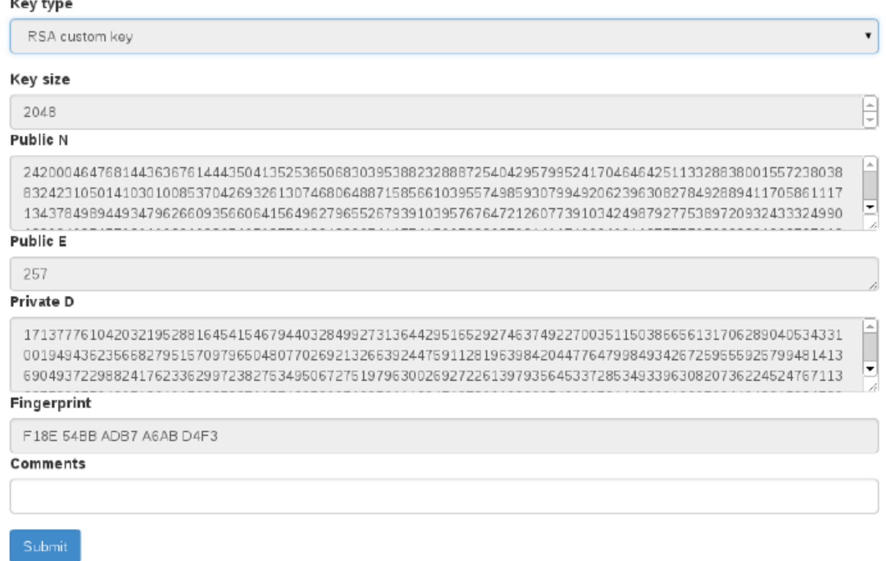


Figura 4.5: Creación de una nueva clave asimétrica

Cuentas

El último módulo del sistema local es **Cuentas**. Este módulo permite manejar las altas, bajas y modificaciones de cualquier cuenta en un servidor externo, y permite asociarlas a las claves públicas existentes. La principal funcionalidad de este módulo consiste en dar a conocer a un servidor, en el transcurso del

proceso de autenticación, cuáles son las claves públicas asociadas a la cuenta en ese sitio.

Accounts loaded in the system

Account	Site	Comment	Actions
rul	identi.ca	Es una prueba.	
rbenencia	gmail	Un webmail.	
rul	login.kalgan.cc		

Figura 4.6: Listado de cuentas asociadas al sistema

Desde este apartado se puede obtener una visión global de todos las cuentas que la aplicación conoce. Se muestra un listado con los nombres y servidores asociados, y para cada una se provee un enlace donde se puede obtener más información, como la cantidad de claves asociadas. Además, sobre cada cuenta se muestra un pequeño comentario opcional. De forma análoga al módulo de **Claves**, los botones verdes y rojos permiten editar o eliminar cada una de las cuentas.

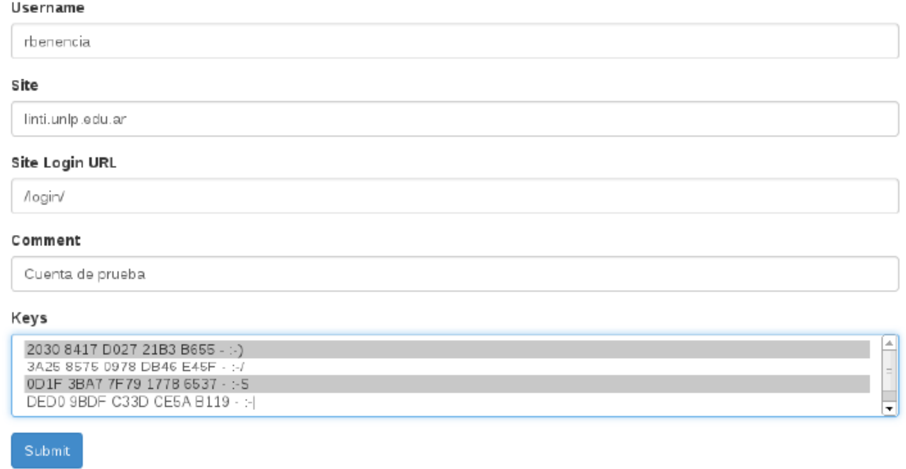


Figura 4.7: Creación de una nueva cuenta

La figura 4.7 muestra la pantalla de creación de una nueva cuenta. Para crear una nueva cuenta se deben proporcionar los siguientes campos de información:

- `Username`: El nombre de usuario con el cual el servidor identifica la cuenta.
- `Site`: El *hostname* del sitio. Cuando el servidor indique quién es, deberá enviar exactamente este nombre, pues desde la aplicación local se hace

una comparación carácter por carácter para buscar cuáles son las cuentas asociadas.

- `Comment`: Comentario descriptivo opcional.
- `Keys`: Este campo permite asociar a la nueva cuenta una o más claves públicas. Para ello, se muestra un listado con los *fingerprints* de las claves disponibles.

4.1.2. *Backend* de la aplicación

Llamamos *frontend* de la aplicación a todo aquello con lo que el usuario puede interactuar directamente. Por el contrario, el *backend* de la aplicación es todo aquello que se ejecuta sin intervención directa del usuario.

Internamente, la aplicación posee dos componentes principales:

- Base de datos
- Comunicación entre cliente y servidor

Base de datos

La base de datos se encarga de almacenar las claves públicas y privadas del usuario, las cuentas del usuario, y las asociaciones entre éstas. También guarda un registro de todos los eventos de importancia, como el alta, baja o modificación de las claves y cuentas, la petición de autorización por parte de un sitio externo, entre otros.

La base de datos se compone de las tablas `account`, `key_pair`, `key_account` y `log`. Las dos primeras almacenan las cuentas y claves asimétricas del usuario, mientras que `key_account` almacena las relaciones entre éstas. Con el objetivo de ilustrar la estructura interna de la base de datos las siguientes figuras muestran sencillas consultas que interactúan con ella.

```
> select * from account limit 1;
id          username    site        site_u_r_l  comment
----------  ----------  ----------  ----------  --------------
1           rul         identi.ca               Es una prueba.
```

Figura 4.8: Consulta en la tabla `account`

La figura 4.8 muestra una consulta sobre la tabla `account`. En ella se puede ver que las columnas de la tabla son `id`, que simplemente es la clave auto-

incremental de la base de datos, `username`, que guarda el nombre de usuario de la cuenta, `site`, que identifica al sitio donde se realizará la autenticación, y `comment`, que contiene un comentario descriptivo opcional de la cuenta.

La columna `site_u_r_l`, que hace referencia a "site URL", se reserva para usos futuros de la aplicación, donde se permitirá que el usuario se autentique con cualquier sitio simplemente desde el *frontend* de la aplicación. Este campo contendrá la URL del sitio que permita utilizar este tipo de autenticación.

```
> select id,type,comment,size,fingerprint from key_pair limit 1;
id     type          comment    size   fingerprint
-----  ------------  --------   -----  ------------------------
1      RSACustomKey  :-|        2048   DED0 9BDF C33D CE5A B119
```

Figura 4.9: Consulta en la tabla `key_pair`

La figura 4.9 muestra una consulta sobre la tabla `key_pair`. En ella se puede ver que las columnas principales de la tabla son `comment`, que almacena un comentario descriptivo opcional sobre la clave, `size`, que almacena el tamaño de las claves, y `fingerprint`, que almacena el *fingerprint* de la clave pública. En la figura no se incluyeron las columnas `public_d`, `public_e` ni `private_d` por cuestiones de presentación, pero estas simplemente contienen el valor en base diez de los números que representan.

Se podría argumentar que las columnas `size` y `fingerprint` contienen información redundante, pues ambos valores se calculan a partir de la clave pública. Sin embargo, se eligió almacenarlos en la base de datos por una cuestión de eficiencia, ya que calcular ambos valores una y otra vez para cada requerimiento que se recibe sobre la clave genera *overhead* innecesario.

Finalmente, la columna `type` se reserva para usos futuros de la aplicación, ya que se permitirá manejar varios tipos de claves asimétricas, y de esta forma se podrá variar entre las alternativas disponibles al momento. Esto permitiría, por ejemplo, utilizar juegos de claves asimétricas existentes en otros ámbitos, como, por ejemplo, claves SSH o GPG[26]. Por lo tanto, esta columna describirá qué tipo de clave asimétrica se va a utilizar.

```
> select * from key_account limit 1;
id          key_pair    account
----------  ----------  ----------
2           3           2
```

Figura 4.10: Consulta en la tabla `key_account`

La figura 4.10 muestra una consulta sobre la tabla `key_account`. Es una tabla

sencilla que simplemente almacena las relaciones entre claves y cuentas, por lo que sus columnas son `key_pair`, que contiene el `id` de una entrada en la tabla con el mismo nombre, y `account`, que de forma análoga contiene el `id` de una cuenta.

```
> select * from log limit 2;
id          key_pair    type        time
----------  ----------  ----------  ------------------------------
1           6           KeyUpdated  2014-02-13 20:07:24.404287 UTC
2                       LoginAttem  2014-02-13 20:08:17.552349 UTC
```

Figura 4.11: Consulta en la tabla `log`

Finalmente, la figura 4.11 muestra una consulta sobre la tabla `log`. En ella se puede ver que las principales columnas de esta tabla son `time`, que contiene en formato UTC la hora del registro, `type`, que identifica el tipo de registro (es decir, si fue una alta, baja o modificación de alguna cuenta o clave, o una petición de autenticación), y `key_pair`, que opcionalmente almacena el `id` de una entrada en la tabla de claves. Esto permite al *frontend* de la aplicación generar los enlaces correspondientes en cada uno de los registros mostrados.

Como motor de bases de datos por defecto se utilizó SQLite[1]. SQLite permite utilizar un archivo como base de datos auto-contenida, evitando al usuario la necesidad de configurar un servidor más grande y complejo como MySQL o PostgreSQL. Además, SQLite es transaccional, por lo que cada cambio y consulta a la base de datos es atómico, aislado, consistente y persistente. Por último, aunque no menos importante, el código de SQLite es de dominio público, lo que lo hace perfecto para integrarlo con el código de este trabajo.

Debido al *framework* y lenguaje de programación utilizado, que serán explicados más adelante, el cambio de motor de base de datos es trivial, y se reduce simplemente a modificar un archivo de configuración con los valores requeridos por el motor elegido.

Comunicación entre cliente y servidor

Cuando un cliente se quiere autenticar con un servidor, el servidor se comunicará con el módulo de autenticación HTTP del cliente para establecer el canal de comunicación. Como se explicó en el capítulo anterior, el sistema propuesto está diseñado para que cada módulo defina la forma de establecer el canal de comunicación que mejor se ajuste a las características del protocolo subyacente. En este apartado se explicará el método elegido y desarrollado para el módulo

de autenticación HTTP.

En primer lugar, se debe notar que la aplicación que corre del lado del cliente es web, lo que facilita la implementación del módulo de autenticación puesto que el cliente y el servidor se manejan con el mismo protocolo. Así, pues, la forma de comunicación elegida fue establecer un *websocket*[8] entre ambas partes con el objetivo de permitir la comunicación bidireccional.

El servidor *websocket* escucha del lado del cliente (de esta forma se invierte, si se quiere, el rol de cliente y servidor durante el proceso de autenticación). En principio esto puede parecer contraproducente puesto que, tal vez, se piense que si hay un *firewall* de por medio el inicio de la comunicación se puede ver frustrado. Sin embargo, el módulo está diseñado para que las conexiones hacia el servidor *websocket* se realicen desde Javascript. De esta forma, quien quiere autenticarse ejecutará, naturalmente de forma local, el código Javascript descargado desde el servidor donde se realizará la autenticación, y localmente se establecerá el *websocket* correspondiente. El código Javascript deberá incluir, además, una forma de comunicarse con su servidor. La forma de lograr esto queda en manos de quien implemente este tipo de autenticación del lado del servidor. Se podría optar, por ejemplo, por otro *websocket* o, más sencillo aún, requerimientos AJAX[11].

Establecido el canal de comunicación entre las partes, resta comentar qué formato se utilizó para implementar el protocolo y enviar los mensajes. Recordar que un *websocket* es simplemente un *socket* encapsulado sobre HTTP, y por lo tanto luego de establecerse la conexión se enviarán simplemente bytes. Como es de esperar, la interpretación que se le da a cada byte depende enteramente de la aplicación. Dicho esto, se optó por utilizar JSON[5] debido a su fuerte conexión con el lenguaje Javascript.

La figura 4.12 muestra el envío de mensajes que se realiza cuando el servidor requiere autenticar a un usuario. Se decidió utilizar una semántica de activación de eventos como formato de los mensajes JSON. Es decir, una vez establecido el *websocket*, las partes envían objetos JSON activando eventos en su contraparte. Por ejemplo, el primer mensaje que se envía es del servidor al usuario. Este mensaje genera el evento `AuthRequested` en la aplicación del usuario. El único parámetro que se envía en este evento es `site`, que especifica quién es el servidor que requiere autenticación. Notar que este parámetro es fácilmente *spoofeable*, pero se desmerece esta cuestión por los motivos expuestos en la página 30. Cuando la aplicación del usuario detecta que se activó el evento `AuthRequested`, comprueba que el parámetro recibido, `site`, se corresponda con alguna de las cuentas que tiene almacenada en su base de datos. En caso

```
Servidor a usuario
 {
   "AuthRequested": {
     "site":"login.kalgan.cc"
   }
 }

Usuario a servidor
 {
   "AuthAcknowledged": {
     "user":"rul",
     "key_fingerprint":"2030 8417 D027 21B3 B655"
   }
 }

Servidor a usuario
 {
   "ChallengeReceived": {
     "challenge":"lt396kqzADsiZatJEqT+nijXUaQrE(...)"
   }
 }

Usuario a servidor
 {
   "ChallengeSolved": {
     "challenge_solution":"8cdmStqzg-"
   }
 }
```

Figura 4.12: Ejemplo de implementación del protocolo de comunicación

afirmativo, buscará el nombre de usuario y la clave pública que esté asociada a esa cuenta, y enviará estos datos al servidor en un evento `AuthAcknowledged`.

Cuando el servidor detecta que se activó el evento `AuthAcknowledged`, comprueba que sus parámetros, `user` y `key_fingerprint`, se correspondan con la información que tiene en su base de datos. Es decir, se comprueba que exista un usuario con ese nombre y además, que el *fingerprint* recibido se corresponda con el de alguna de las claves públicas asociadas a esa cuenta. En caso afirmativo, se generará una contraseña de un solo uso y se le asignará un tiempo de expiración. La contraseña se cifrará con la clave pública determinada previamente, y se enviará al usuario el evento `ChallengeReceived` junto con el *challenge* creado. Notar que debido a que el cifrado de la contraseña genera un conjunto de bytes que puede no tener representación en el conjunto de caracteres ASCII[2], se decidió codificar el *challenge* en base 64[25].

Cuando la aplicación del usuario detecta que se activó el evento `ChallengeReceived`, decodificará el parámetro `challenge`, codificado en base 64, y lo descifrará utilizando la clave privada correspondiente a la clave pública previamente seleccionada. Luego, activará el evento `ChallengeSolved` en el servidor.

Cuando el servidor detecta que se activó el evento `ChallengeSolved`, comprobará que el parámetro `challenge_solution` se corresponda con la contraseña generada previamente y, además, que el tiempo de expiración no haya caducado. Si se cumplen ambas condiciones, entonces el servidor habrá autenticado al usuario.

4.1.3. Sobre la elección del lenguaje

La aplicación fue desarrollada utilizando el lenguaje de programación Haskell[22]. Haskell es un lenguaje funcional puro, lo que significa que sus programas están constituidos únicamente por funciones. El concepto de "función" no es el mismo que se utiliza en los lenguajes imperativos, donde una función no es más que una agrupación de sentencias que, tal vez, reciben algún parámetro y devuelven un valor, si no que se asemeja a la definición aritmética, ya que no se manejan datos mutables o de estado. Esta característica, denominada **transparencia referencial**[37], trae importantes consecuencias, siendo una de ellas que el resultado de la invocación a una función depende únicamente de sus parámetros y subexpresiones, y no de factores externos como, por ejemplo, variables globales. La importancia de la transparencia referencial en el ámbito de la informática reside en que permite tanto al programador como al compilador

razonar y extraer conclusiones lógicas sobre el comportamiento de un programa. Esto permite mejorar diversos aspectos de un programa, simplificarlo de forma automática y realizar optimizaciones como *memoization*[28], CSE[3] (*Common Subexpression Elimination*), evaluación *lazy*[40] e incluso paralelización.

Otro aspecto de interés del lenguaje es su sistema de tipos. Haskell tiene tipado estático y fuerte, lo que implica que se buscarán errores de tipo (por ejemplo, que una función reciba únicamente parámetros del tipo para el cual fue programada) en tiempo de compilación. Además, Haskell utiliza inferencia de tipos, lo cual permite al programador escribir funciones sin especificar el tipo de sus parámetros ni valor de retorno. Esto es posible gracias a que el lenguaje analiza con qué operadores y operandos son utilizados los valores del programa, y en base a sus conclusiones determina unívocamente el tipo de datos del valor. Cuando se presentan ambigüedades porque no hay información suficiente para determinar el tipo de datos, entonces el programador deberá especificarlo manualmente.

Figura 4.13: El logotipo del lenguaje Haskell

No existe una razón particular por la cual se eligió Haskell por sobre otros lenguajes para realizar este desarrollo, más que la preferencia del autor. Sin embargo, es importante destacar que, debido a que Haskell es un lenguaje compilado y no interpretado, la distribución del programa se facilita puesto que consiste de un único archivo binario con un par de archivos de configuración. Además, puesto que se desarrolló una aplicación web es natural imaginar que se precisará la instalación y configuración de un servidor web para hacer uso de ésta. Este requerimiento es un claro obstáculo para un usuario final, quien probablemente no disponga de los conocimientos para realizar este procedimiento. Por el contrario, Haskell tiene la interesante característica de permitir importar un servidor web como una librería más[36]. Esto también es posible hacerlo en otros lenguajes, como Python o Ruby, pero a diferencia de éstos, Haskell compilará tanto la aplicación como el servidor web en un único archivo binario. Esta facilidad trae la importante ventaja de permitir la distribución de la aplicación sin necesidad de instalar librerías adicionales.

4.1.4. El *framework* Yesod

El uso de *frameworks* facilita de forma considerable el desarrollo de aplicaciones, eliminando la necesidad de programar código que tiende a ser común en el ámbito para el cual el *framework* fue desarrollado. En el ámbito web ha emergido el concepto de *Web Application Framework*[41] que, como el nombre sugiere, caracteriza a los *frameworks* orientados al desarrollo de aplicaciones web. Esta clase de *frameworks* permite aliviar al programador de la necesidad de programar código común en aplicaciones web, como el acceso a base de datos, el uso de *templates* para crear vistas, código para el manejo de sesiones, entre otros.

Haskell dispone de una variada cantidad *frameworks* enfocados a distintas áreas[21]. En el ámbito web los principales *frameworks* disponibles son Happstack[19], Snap[4] y Yesod[42]. Los tres son muy interesantes, de libre distribución, y disponen de una buena comunidad de usuarios y desarrolladores. Para realizar la aplicación de este trabajo de grado se decidió utilizar Yesod puesto que la documentación es muy completa e incluso dispone de un libro de libre acceso[35].

Yesod intenta reducir la complejidad del desarrollo de aplicaciones web mediante las fortalezas de Haskell. Las facilidades de este lenguaje permiten encontrar y eliminar errores en tiempo de compilación. La transparencia referencial, concepto introducido en la sección anterior, asegura que la aplicación no contendrá *side effects*[23]. Yesod utiliza el sistema de tipos de Haskell para erradicar clases enteras de *bugs* como XSS[15] (*Cross Site Scripting*) o inyecciones SQL[16]. También aprovecha este sistema para generar estáticamente URLs dentro de la aplicación, asegurando de esta forma la ausencia de enlaces rotos.

```
/keys                    KeyPairR           GET
/keys/new                KeyPairNewR        GET POST
/keys/new/generate       KeyPairGenerateR   GET
/key/#KeyPairId/edit     KeyPairEditR       GET POST
/key/#KeyPairId/qr       KeyPairQRCodeR     GET
/key/#KeyPairId/delete   KeyPairDeleteR     POST
/key/#KeyPairId          KeyPairDetailR     GET

/accounts                    AccountR       GET
/accounts/new                AccountNewR    GET POST
/account/#AccountId/edit     AccountEditR   GET POST
/account/#AccountId/delete   AccountDeleteR POST
/account/#AccountId          AccountDetailR GET
```

Figura 4.14: Configuración de rutas en Yesod

La figura 4.14 muestra un fragmento de la configuración de las rutas de la aplicación desarrollada. La sintaxis de cada línea consiste en la tríada URL, *handler* y método HTTP. La URL especifica el formato que deberá tener una URL para ser derivada a un *handler*. Por ejemplo, para exportar un juego de claves asimétricas a una imagen con un código QR, el usuario deberá acceder a la URL `/key/#KeyPairId/qr`. Los fragmentos de una URL que comienzan con el símbolo `#` definen un parámetro y especifican su tipo. En este caso, el único parámetro de la URL tiene el tipo de dato `KeyPairId`, que representa una clave primaria de la entidad `KeyPair` en la base de datos. De esta forma, cuando el usuario realice un requerimiento sobre esta URL, Yesod se encargará automáticamente de aprovechar el sistema de tipos de Haskell e interpretar el parámetro con el tipo ya mencionado. Si la interpretación falla porque, por ejemplo, se intentó realizar una inyección SQL, automáticamente se devolverá un error al usuario.

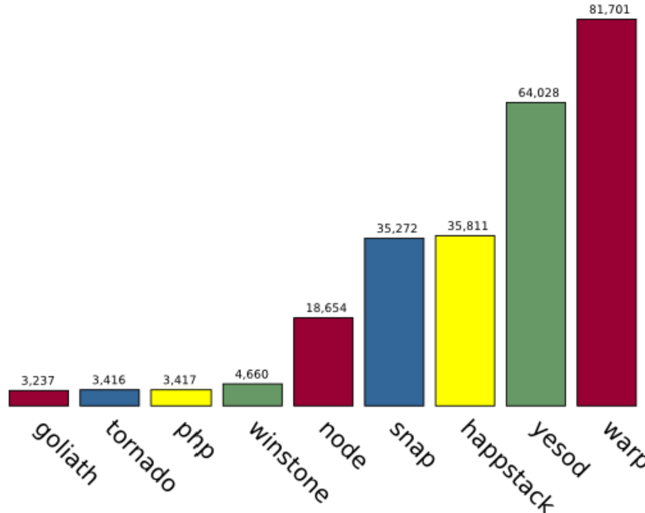


Figura 4.15: *Benchmarks* de aplicaciones web y servidores de aplicaciones web

Por último, aunque no menos importante, se eligió Yesod como *framework* para el desarrollo por su sencilla integración con Warp[36], un servidor web escrito en Haskell. Esta combinación de tecnologías, que lamentablemente no tiene la difusión que debería, tiene la importante ventaja de ser altamente eficiente en el manejo de recursos. La figura 4.15 muestra un gráfico de barras con la cantidad de requerimientos por segundos que soportan distintas tecnologías, con Yesod y Warp a la cabeza [1].

[1] El gráfico fue realizado por Michael Snoyman, y posee licencia Creative Commons Attribution 4.0. Para obtener más información sobre los *benchmarks*, visitar: `http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks`

4.2. Servidor de ejemplo

Como se ha mencionado anteriormente, para que la prueba de concepto del sistema de autenticación presentado esté completa, se programaron tres componentes. El primero se explicó en la sección anterior, y consiste en un software que corre en el dispositivo del usuario y que permite almacenar los juegos de claves y administrar los *challenges* recibidos.

La segunda aplicación desarrollada consiste, por supuesto, en una aplicación web que implemente el sistema de autenticación propuesto.

4.2.1. Funcionalidad de la aplicación

La aplicación desarrollada, sencilla debido a que es una simple prueba de concepto, consiste en una página inicial, mostrada en la figura 4.16, con un enlace a una página restringida. Esta página restringida es accesible solo por los usuarios que han ingresado al sistema.

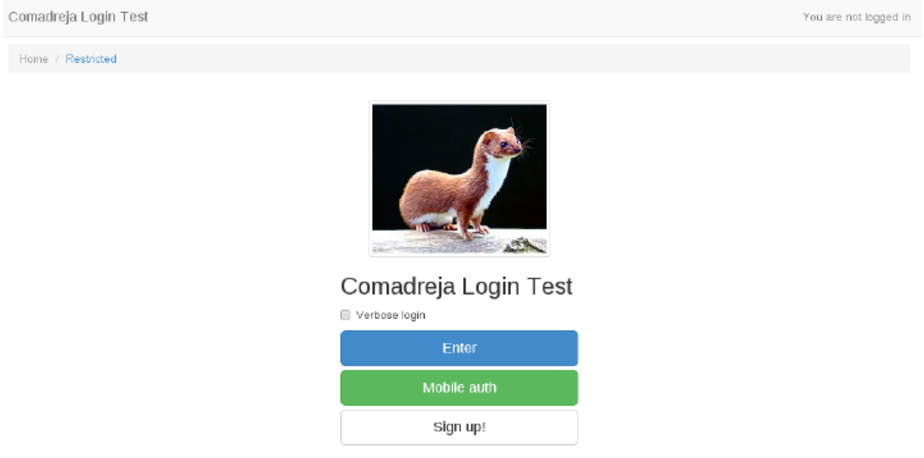


Figura 4.16: Captura de pantalla de la página inicial

Naturalmente el sitio posee una sección donde el usuario deberá registrar su cuenta y asociarle una clave pública. Este comportamiento se aprecia en la figura 4.17.

Es innegable que la asociación entre una cuenta y una clave pública se puede realizar de forma más sencilla que el ingreso manual de los campos, pero debido a que este sitio es simplemente una prueba de concepto, no se ha trabajado en ese aspecto y se propone como una línea de investigación futura.

Username

Your username...

Public key size

Enter the size of the key in bits...

Public key number

Enter public key...

Public key exponent

Enter the exponent used to generate your public key...

Public key fingerprint

Enter the fingerprint shown in your Comadreja instance...

Sign up!

Figura 4.17: Captura de pantalla de la página de registro

Luego de que el usuario haya registrado una cuenta, asociado una clave pública, y ejecutado una instancia de la aplicación local descripta en la sección anterior, el proceso de autenticación se podrá realizar simplemente mediante el botón **Enter** (ver figura 4.16). Detrás de escena se establecerá un *websocket* y se intercambiarán los mensajes JSON ya mencionados, y si los datos resultantes son correctos entonces la autenticación se habrá efectuado de forma exitosa, y el usuario podrá acceder al contenido restringido. De esta forma se obtiene como resultado el objetivo planteado en este trabajo de grado, que es ofrecer un sistema de autenticación con beneficios similares a los contemporáneos, como OpenID, pero sin sacrificar por ello la privacidad del usuario (recordar que un proveedor de OpenID conoce cuándo y dónde se autentica el usuario).

Finalmente, si el usuario quiere autenticarse desde un dispositivo donde no tiene asociada una clave privada pero tiene consigo un dispositivo móvil con cámara previamente asociado, como un teléfono, entonces podrá autenticarse en la aplicación mediante el botón **Mobile auth**. Al ingresar en esta opción la aplicación ofrecerá un campo de entrada de texto donde el usuario deberá escribir su nombre de usuario. Luego, el sistema creará una contraseña temporal y la cifrará con la clave pública asociada a esa cuenta, creará un objeto JSON que incluirá el *fingerprint* de la clave utilizada y el *challenge* codificado en base 64[25], para luego codificarlo en un código QR[24] que finalmente será leído e interpretado por el teléfono del usuario. La figura 4.18 muestra una captura de pantalla con un código QR generado desde la aplicación.

Figura 4.18: Captura de pantalla del código QR de un *challenge*

Luego, el usuario podrá descifrar el *challenge* desde su teléfono utilizando una herramienta específicamente desarrollada para este propósito, y escribir la contraseña en el campo de texto correspondiente.

4.2.2. Sobre del desarrollo de la aplicación

La aplicación que implementa el sistema de autenticación propuesto fue desarrollada utilizando el lenguaje de programación PHP debido a su idoneidad para construir sistemas web. La persistencia de los usuarios y claves públicas se logra utilizando una base de datos SQLite[1] que contiene únicamente dos tablas, `publickey` y `user`. Se utilizó la librería RedBean[30] como interfaz entre la base de datos y PHP, debido a su facilidad y simplicidad de configuración.

La comunicación vía *websocket* con el cliente se programó utilizando Javascript, y la comunicación por parte del cliente desde el código Javascript al servidor se programó utilizando requerimientos AJAX.

Ambas partes del desarrollo son de libre distribución, y se encuentran bajo la licencia GNU General Public License[13] versión 3.

4.3. Aplicación para dispositivos móviles

El último componente desarrollado para el presente trabajo de grado consiste en una aplicación para teléfonos con sistema operativo Android que permite importar claves privadas y decodificar *challenges* requeridos por un servidor.

La aplicación es lo suficientemente sencilla como para ser fácilmente portada a otras plataformas móviles.

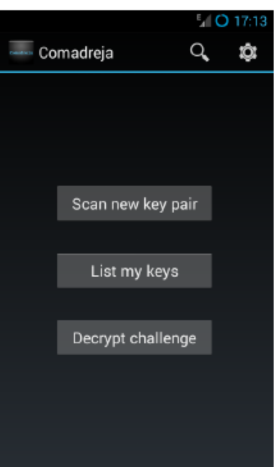


Figura 4.19: Menú principal de la aplicación para teléfonos

La figura 4.19 muestra el menú inicial de la aplicación móvil. Este menú permite al usuario elegir importar un nuevo juego de claves asimétricas, listar las claves existentes y descifrar *challenges*.

La importación de claves se realiza mediante la cámara del dispositivo, utilizando la funcionalidad del cliente local descripta en la figura 4.3. El formato del mensaje enviado por dicha aplicación consiste en una sencilla imagen con un código QR que contiene un objeto JSON que a su vez contiene el *fingerprint* de la clave pública utilizada para realizar el cifrado, junto con el *challenge* codificado en base 64.

Cuando un usuario desee autenticarse de forma móvil en un dispositivo que no tiene una clave privada asociada, podrá utilizar para tal fin su teléfono móvil solo si previamente exportó un juego de claves a la aplicación del teléfono y asoció la correspondiente clave pública al sitio donde se requiera la autenticación. Si se cumplen ambas condiciones, el usuario podrá pulsar *Decrypt challenge* para activar la cámara del dispositivo, decodificar el código QR generado y descifrar el *challenge*. Cuando esta operación ocurre de forma exitosa, la contraseña elegida por el servidor aparecerá en pantalla y el usuario deberá introducirla manualmente en el dispositivo correspondiente. Si el tiempo de expiración asignado a la contraseña no caducó al momento de su ingreso, entonces el usuario se habrá autenticado correctamente.

Por la naturaleza del sistema operativo donde se ejecuta, la aplicación se desarrolló con el lenguaje de programación Java utilizando el *framework* que posee

Android para controlar el sistema operativo. Para lograr la persistencia de las claves asimétricas importadas a la aplicación se utiliza el motor de base de datos *SQLite*[1][39], debido a que es el mecanismo elegido por el sistema operativo para que las aplicaciones almacenen su información.

La aplicación es de libre distribución, y se encuentra bajo la licencia GNU General Public License[13] versión 3.

Conclusión

El objetivo teórico de este trabajo ha sido proponer un mecanismo de autenticación donde las credenciales se almacenen de forma descentralizada utilizando criptografía asimétrica.

Para tal fin, se han investigado las principales tecnologías de autenticación con especificaciones abiertas y capacidades de *Single Sign-On*. Se encontró que la tendencia actual en este ámbito es utilizar los llamados "*logins* sociales", que permiten a sus usuarios autenticarse en distintos sitios utilizando *Single Sign-On* mediante una red social. Sin embargo, no hay una especificación que estandarice una metodología para desarrollar un sistema de autenticación de este tipo, y en general se implementa como un híbrido entre dos tecnologías también analizadas, OpenID para realizar autenticación de forma federada, y OAuth como *framework* de autorización. OpenID es una tecnología que permite identificar a un usuario mediante una URL. De esta forma, los usuarios pueden elegir un proveedor de OpenID, o incluso administrar uno propio, y utilizarlo para autenticarse. El auge de OpenID ocurrió en el año 2008 cuando surgieron varios proveedores y consumidores de esta tecnología, pero pronto vio su declive en pos de otras formas de autenticación. Por otro lado, OAuth surgió de la necesidad de compartir funcionalidad entre distintas aplicaciones web sin por ello tener que confiar las credenciales de acceso a una entidad externa. OAuth ha progresado hacia su segunda especificación, OAuth 2.0, y actualmente es utilizado como tecnología base de los *logins* sociales para proveer tanto autorización como autenticación. Finalmente, en un intento de devolver federalización a los usuarios, el grupo de desarrolladores de OpenID ha publicado en febrero de 2014 la especificación de OpenID Connect, que se apoya sobre OAuth para

proveer una capa adicional de identidad.

Se detectaron tres problemas importantes en el paradigma de autenticación analizado. En primer lugar, se considera desfavorable que las credenciales estén centralizadas en un servidor. Esto es una práctica habitual en aplicaciones web, ya que generalmente mantienen una base de datos de sus usuarios junto con sus contraseñas. No se critica el aspecto técnico de la centralización, sino el impacto que produciría el compromiso del servidor que contiene las credenciales.

El segundo problema detectado en el paradigma actual de la autenticación por Internet consiste en la homogeneidad de las credenciales de los usuarios, que por comodidad o dificultad en recordar muchas contraseñas tienden a repetirlas o utilizar patrones predecibles para generarlas, y por lo tanto el compromiso de uno de los servidores donde el usuario tenga una cuenta implicaría un probable compromiso del resto de sus cuentas con credenciales similares. Análogamente, en sistemas de autenticación con capacidades de *Single Sign-On*, como OpenID, el compromiso de la contraseña utilizada implicaría automáticamente el compromiso del total de las cuentas asociadas a la identidad.

El último problema detectado consiste en la confianza que debe depositar un usuario en un proveedor de *Single Sign-On*. Utilizando las tecnologías de autorización y autenticación actuales, el proveedor de identidad sabrá en qué sitios y en qué momento el usuario ingresa a otros servicios, lo cual constituye una importante pérdida de privacidad. Además, aún si las tecnologías con especificaciones abiertas como OpenID permiten a sus usuarios administrar su propia entidad de autenticación, el conocimiento para hacer esto no está al alcance ni es del interés de todos los usuarios.

Luego de analizar los problemas se planteó una infraestructura de trabajo que permite resolverlos. El problema de la centralización de credenciales fue resuelto utilizando criptografía asimétrica. La propuesta es que el usuario asocie claves públicas a sus cuentas, y que conserven de forma separada la clave privada correspondiente. De esta manera, el servidor no tiene la información de acceso centralizada, si no que está distribuida entre cada usuario.

El problema de la homogeneidad de credenciales se solucionó, también, utilizando las propiedades de la criptografía asimétrica. Para ello, se propuso que el usuario pueda crear y asociar fácilmente nuevos juegos de claves asimétricas sin por ello tener que recordar nueva información. De esta forma se puede obtener funcionalidad similar a la de *Single Sign-On* (manteniendo las claves asimétricas simétricamente cifradas, y descifrándolas una única vez cuando, por ejemplo, el usuario inicia sesión en su sistema operativo), pero sin utilizar la misma credencial en los sitios asociados.

Por último, el problema de la confianza obligada sobre una entidad externa se solucionó proponiendo la utilización de un software del lado del cliente que permita al usuario almacenar de forma segura sus claves privadas e interactuar con los sitios que requieran autenticación. De esta forma la comunicación entre quien quiere autenticar y quien quiere ser autenticado es bidireccional, sin requerir la presencia de un tercero.

Luego de exponer la necesidad de una aplicación que corra del lado del cliente, se propuso una posible arquitectura de software que cumple con los requisitos expuestos. El software debe poseer varios componentes definidos. En primer lugar se precisa un "Depósito de claves", que almacenará de forma segura las claves privadas del usuario. También se precisa un "Manejador de *challenges*", que se encargará de descifrar y responder los *challenges* recibidos. Estos *challenges* se envían desde los "Módulos de autenticación", que son el tercer componente de la infraestructura y se encargan de establecer la comunicación con los servidores que requieren autenticación. La comunicación se da mediante un "Protocolo" común, también propuesto como parte de la infraestructura.

Con el objetivo de eliminar la limitación de poder autenticarse únicamente mediante dispositivos que dispongan de una clave privada asociada, se propuso un mecanismo para utilizar dispositivos móviles, como teléfonos celulares, para asociarles claves privadas mediante códigos QR, y además permitir descifrar *challenges* utilizando la misma metodología. De esta forma, el usuario podrá ingresar manualmente la contraseña descifrada en cualquier dispositivo con el cual se quiera autenticar, sin la necesidad de que éste tenga una clave privada asociada.

Finalmente, luego de plantear la necesidad de un software dedicado y proponer la arquitectura de su funcionamiento, se desarrolló una prueba de concepto que permite almacenar y generar juegos de claves asimétricas, y recibir y decodificar *challenges* propuestos por aplicaciones web. También se desarrolló una prueba de concepto de un sitio web que implementa este tipo de autenticación, y una aplicación para dispositivos móviles que permite importar claves privadas y descifrar *challenges*. Los tres desarrollos fueron liberados bajo la licencia GNU GPL, y por lo tanto son de libre distribución.

Así pues, se ha desarrollado un mecanismo de autenticación que no centraliza credenciales, fomenta el uso de claves distintas para sitios distintos y, por último, no depende de una entidad externa para realizar el proceso de autenticación. Se reconoce que la adopción masiva de este sistema se dificulta debido la complejidad inherente de requerir un software adicional que corra del lado del cliente. Sin embargo, existen ambientes donde su utilización es propicia. Por

ejemplo, ambientes de desarrollo, donde es de suponer que sus usuarios poseen conocimiento técnico, o ambientes donde se precise un *hardware token* para realizar la autenticación, ya que éste podría ser reemplazado por el mecanismo desarrollado.

5.1. Trabajo futuro

Como trabajo a futuro se propone, en primer lugar, desarrollar módulos que implementen el sistema de autenticación propuesto en distintas plataformas libres, por ejemplo, gestores de contenidos como Joomla o Drupal, plataformas educativas como Moodle, entre otros. Avanzar hacia la adopción de un sistema de autenticación con estas características requiere que este mecanismo esté disponible en una importante cantidad de sitios, ya que de esta forma se justifica su uso.

Como segunda línea de investigación se propone que el sistema y protocolo desarrollado permitan utilizar distintos algoritmos de cifrado. Actualmente se están utilizando claves asimétricas RSA en crudo, es decir, sin ningún formato en particular. Sin embargo, no sería complicado extenderlo para que utilice claves GPG o incluso claves SSH. Si un sistema con estas características tiene como precepto en su diseño el hecho de soportar distintos algoritmos de cifrado, en caso de que se descubriera una vulnerabilidad en uno de ellos y se volviera inservible, sería trivial migrar a otro que no lo sea.

Finalmente, una última propuesta es trabajar en la automatización de la asociación de cuentas entre el sistema que se ejecuta localmente y los sitios que utilizan el mecanismo de autenticación propuesto. Actualmente, para que el sistema funcione el usuario debe ingresar la información manualmente. Sin embargo, no sería difícil establecer una convención o API entre cliente y servidor que permita establecer la asociación de forma automática.

Bibliografía

[1] *About SQLite*. URL: https://sqlite.org/about.html (visitado 08-04-2014).

[2] V.G. Cerf. *ASCII format for network interchange*. RFC 20. Internet Engineering Task Force, oct. de 1969. URL: http://www.ietf.org/rfc/rfc20.txt.

[3] John Cocke. "Global Common Subexpression Elimination". En: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, 1970, págs. 20-24. DOI: 10.1145/800028.808480. URL: http://doi.acm.org/10.1145/800028.808480.

[4] Gregory Collins y Doug Beardsley. "The Snap Framework: A Web Toolkit for Haskell". En: *Internet Computing, IEEE* 15.1 (2011), págs. 84-87.

[5] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). Obsoleted by RFCs 7158, 7159. Internet Engineering Task Force, jul. de 2006. URL: http://www.ietf.org/rfc/rfc4627.txt.

[6] W. Diffie y M. Hellman. "New Directions in Cryptography". En: *IEEE Trans. Inf. Theor.* 22.6 (sep. de 2006), págs. 644-654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[7] Taher El Gamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". En: *Proceedings of CRYPTO 84 on Advances in Cryptology*. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1985, págs. 10-18. ISBN: 0-387-15658-5. URL: http://dl.acm.org/citation.cfm?id=19478.19480.

[8] I. Fette y A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, dic. de 2011. URL: `http://www.ietf.org/rfc/rfc6455.txt`.

[9] R. Fielding y col. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Internet Engineering Task Force, jun. de 1999. URL: `http://www.ietf.org/rfc/rfc2616.txt`.

[10] J. Galbraith, J. Van Dyke y J. Bright. *Secure Shell Public Key Subsystem*. RFC 4819 (Proposed Standard). Internet Engineering Task Force, mar. de 2007. URL: `http://www.ietf.org/rfc/rfc4819.txt`.

[11] J. J. Garrett. *Ajax: A New Approach to Web Applications*. Adaptive Path LLC, http://www.adaptivepath.com/publications/essays/archives/000385.php. 2005.

[12] GCHQ. *Public-Key Encryption - how GCHQ got there first!* 1997. URL: `http://archive.is/wd6Ff` (visitado 07-02-2014).

[13] *GNU General Public License*. Inglés. Versión 3. Free Software Foundation, 29 de jun. de 2007. URL: `http://www.gnu.org/licenses/gpl.html`.

[14] *Google OpenID Connect compliance*. URL: `https://developers.google.com/accounts/docs/OAuth2Login#oidc-compliance` (visitado 29-04-2014).

[15] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

[16] WG Halfond, Jeremy Viegas y Alessandro Orso. "A classification of SQL-injection attacks and countermeasures". En: *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA*. 2006, págs. 13-15.

[17] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849 (Informational). Obsoleted by RFC 6749. Internet Engineering Task Force, abr. de 2010. URL: `http://www.ietf.org/rfc/rfc5849.txt`.

[18] Eran Hammer-Lahav. *Beginner's Guide to OAuth*. 2010. URL: `http://oauth.net/core/1.0/` (visitado 07-04-2014).

[19] *Happstack*. URL: `http://happstack.com/page/view-page-slug/1/happstack` (visitado 22-04-2014).

[20] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, oct. de 2012. URL: `http://www.ietf.org/rfc/rfc6749.txt`.

[21] *Haskell Frameworks*. URL: `http://www.haskell.org/haskellwiki/Web/Frameworks` (visitado 22-04-2014).

[22] P. Hudak y J. Fasel. "A Gentle Introduction to Haskell". En: *ACM SIGPLAN Notices* 27.5 (mayo de 1992).

[23] John Hughes. "Why functional programming matters". En: *The computer journal* 32.2 (1989), págs. 98-107.

[24] International Organization for Standardization. *Information Technology — Automatic Identification and Data Capture Techniques — QR Code 2005 Bar Code Symbology Specification*. ISO/IEC 18004:2006. 2006.

[25] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648 (Proposed Standard). Internet Engineering Task Force, oct. de 2006. URL: `http://www.ietf.org/rfc/rfc4648.txt`.

[26] Werner Koch. *GnuPG: GNU Privacy Guard*. 1998.

[27] Loren McDonald. *Social Login: A Data Capture Game Changer*. URL: `http://www.silverpop.com/blogs/email-marketing/social-login-data-capture.html` (visitado 29-04-2014).

[28] Donald Michie. "Memo functions and machine learning". En: *Nature* 218.5136 (1968), págs. 19-22.

[29] Victor S Miller. "Use of Elliptic Curves in Cryptography". En: *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*. Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1986, págs. 417-426. ISBN: 0-387-16463-4. URL: `http://dl.acm.org/citation.cfm?id=18262.25413`.

[30] Gabor de Mooij. *RedBeanPHP Easy ORM for PHP*. URL: `http://www.redbeanphp.com/` (visitado 17-04-2014).

[31] *OAuth Core 1.0*. 2007. URL: `http://oauth.net/core/1.0/` (visitado 07-04-2014).

[32] David Recordon y Drummond Reed. "OpenID 2.0: A Platform for User-centric Identity Management". En: *Proceedings of the Second ACM Workshop on Digital Identity Management*. DIM '06. Alexandria, Virginia, USA: ACM, 2006, págs. 11-16. ISBN: 1-59593-547-9. DOI: `10.1145/1179529.1179532`. URL: `http://doi.acm.org/10.1145/1179529.1179532`.

[33] D Reed y D McAlpin. *Extensible Resource Identifier Syntax 2.0, OASIS Committee Specification, OASIS XRI Technical Committee*. 2005.

[34] Nat Sakimura y col. *Openid connect standard 1.0*. 2011.

[35] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., 2012.

[36] Michael Snoyman. "Warp: A Haskell Web Server." En: *IEEE Internet Computing* 15.3 (2011).

[37] Harald Søndergaard y Peter Sestoft. "Referential transparency, definiteness and unfoldability". En: *Acta Informatica* 27.6 (1990), págs. 505-517.

[38] Don Thibeau. *The OpenID Foundation Launches the OpenID Connect Standard*. URL: `http://openid.net/2014/02/26/the-openid-foundation-launches-the-openid-connect-standard/` (visitado 29-04-2014).

[39] Lars Vogel. "Android SQLite Database and ContentProvider-Tutorial". En: *Copyright* 2011 (2010), pág. 2012.

[40] Christopher P Wadsworth. "Semantics and Pragmatics of the Lambda-Calculus." Tesis doct. University of Oxford, 1971.

[41] *Web Application Framework*. URL: `http://docforge.com/wiki/Web_application_framework` (visitado 21-04-2014).

[42] *Yesod*. URL: `http://www.yesodweb.com/` (visitado 22-04-2014).

[43] T. Ylonen y C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251 (Proposed Standard). Internet Engineering Task Force, ene. de 2006. URL: `http://www.ietf.org/rfc/rfc4251.txt`.