

# PROTOCOLO CARONTE

## SERVIDOR DE AUTENTICACIÓN DE USUARIOS Y DISTRIBUCIÓN DE CLAVES

Por: José Aarón López García



*Pintura al Óleo “La barca de Caronte”, de José Benlliure Gil, Valencia, 1919.*



## ***Índice***

|   |    |
|---|----|
| 1. Objetivos y motivación .....                           | 5  |
| 2. Introducción a la autenticación de usuarios .....      | 7  |
| 3. Técnicas de seguridad y autenticación .....            | 9  |
| 4. Tecnologías actuales .....                             | 11 |
| 4.1. Cifrado Asimétrico .....                             | 12 |
| 4.2. Cifrado Simétrico .....                              | 13 |
| 4.3. Protocolo Kerberos .....                             | 14 |
| 5. Ingeniería de Software: diseño y modelado .....        | 19 |
| 5.1. Tabla de requisitos .....                            | 20 |
| 5.2. Tabla de tareas .....                                | 21 |
| 5.3. Diagrama de Gantt .....                              | 22 |
| 5.4. Diagrama de Casos de Uso .....                       | 23 |
| 5.5. Diagrama del protocolo .....                         | 24 |
| 5.6. Diagrama de flujo .....                              | 25 |
| 5.7. Diagrama Entidad-Relación de la Base de Datos .....  | 26 |
| 6. Protocolo Caronte: Introducción .....                  | 27 |
| 7. Principales ataques cibernéticos y contramedidas ..... | 33 |
| 7.1. Man-In-The-Middle .....                              | 34 |
| 7.2. Ataque tipo Replay .....                             | 35 |
| 7.3. Ataque Selfie .....                                  | 37 |
| 7.4. Ataques por fuerza bruta .....                       | 38 |
| 7.4.1. Ataques por diccionario .....                      | 39 |
| 7.4.2. Ataques por fuerza bruta inversa .....             | 41 |
| 7.5. Ataques específicos de Kerberos .....                | 42 |
| 7.6. Ataques de inyección de código .....                 | 44 |
| 8. Proyecto Caronte .....                                 | 45 |
| 8.1. Instalar y ejecutar el proyecto .....                | 46 |
| 8.2. Ejecutar la aplicación de prueba .....               | 48 |
| 8.3. Análisis de la prueba .....                          | 53 |

|  |    |
|--|----|
| 8.4. Principales componentes software .....    | 57 |
| 8.4.1. Estructura interna de los Tickets ..... | 58 |
| 8.4.1.1. Ticket Granting Ticket .....          | 59 |
| 8.4.1.2. Service Granting Ticket .....         | 60 |
| 8.4.1.3. Key Granting Ticket .....             | 61 |
| 8.4.1.4. Fake Ticket .....                     | 62 |
| 8.4.2. Interfaz Criptográfica .....            | 63 |
| 8.4.3. Almacenamiento de credenciales .....    | 65 |
| 8.4.4. Base de Datos del servidor .....        | 67 |
| 8.4.4.1. Tabla del Usuario .....               | 68 |
| 8.4.4.2. Tabla del Ticket .....                | 69 |
| 8.4.4.3. Tabla de la Sesión .....              | 70 |
| 8.4.5. API REST del Servidor .....             | 71 |
| 8.4.5.1. Caronte Authenticator .....           | 72 |
| 8.4.5.2. Ticket Validator .....                | 73 |
| 8.4.5.3. Ejemplo de registro y proveedor ..... | 74 |
| 8.4.6. Código del cliente .....                | 76 |
| 8.4.6.1. Login y generación de tickets .....   | 78 |
| 8.4.6.2. Validación de otros tickets .....     | 79 |
| 9. Conclusión y Bibliografía .....             | 80 |

## **1. Objetivos y motivación**

Uno de los principales problemas a los que se enfrenta todo desarrollador de aplicaciones web es la autenticación de usuarios, control de acceso y seguridad de la plataforma software.

Para una aplicación telemática es importante no solo poder identificar a los usuarios que intentan acceder al servicio, sino también impedir acceso a usuarios sin credenciales así como garantizar una comunicación segura entre el usuario y el servidor, minimizando (o eliminando por completo) cualquier posibilidad de que la transmisión sea interceptada e interpretado por terceros sin permisos.

La criptografía moderna provee todas las herramientas y primitivas necesarias para construir un buen sistema de autenticación. Sin embargo, no existe un estándar o solución única, mucho menos una herramienta especializada de uso general que pueda facilitar la construcción de un mecanismo para identificar usuarios y proteger la información sin que el desarrollador tenga que entrar en grandes detalles sobre ciberseguridad.

Proteger una aplicación web es un trabajo tedioso que suele consumir muchos recursos humanos y la solución suele ser específica a la aplicación en la que se está trabajando. No sólo eso; aunque no existe una herramienta estándar para autenticar usuarios, sí que existen métodos muy conocidos y ampliamente aceptados por la comunidad, lo que se traduce a una gran afluencia de aplicaciones webs que implementan el mismo código (o similar) para realizar la misma tarea. En otras palabras: cada vez que un desarrollador tiene que añadir un login a una nueva plataforma, se suele hacer desde cero, reinventando la rueda.

Sabiendo esto, sería una gran ventaja poder contar con un sistema genérico, reusable y adaptable que permita implementar el proceso de autenticar usuarios independientemente de la aplicación sobre la que se implementa y los datos que se transmiten.

Existen varias soluciones ya estudiadas para resolver este problema, con sus ventajas e inconvenientes. Las estudiaremos en el siguiente apartado.



## 2. Introducción a la autenticación de usuarios

Desde los inicios de las comunicaciones seguras se ha usado siempre un mecanismo de autenticación conocido como *Challenge-Response*.

El mecanismo consiste en que la entidad servidora realiza una “petición” o “pregunta” (el *challenge*), y la entidad cliente proporciona una respuesta correcta (el *response*).

En el *challenge-response* siempre se pide un dato que solo conocerá el cliente que dice ser; una contraseña. Sólo si el cliente realmente conoce esta contraseña podemos asegurar su identidad.

Esta forma simple de autenticación *challenge-response* mediante contraseña se ha usado desde los inicios de Internet. Sin embargo, con la creciente adopción de Internet en todos nuestros aspectos de la vida, así como las mejoras en telecomunicaciones, ha hecho que sea relativamente fácil y abundante encontrar merodeadores por la red cuya única intención es la de observar el tráfico, modificarlo, robar información y demás actividades que ponen en riesgo la seguridad de los usuarios y las plataformas web.

¡Enviar una contraseña como respuesta a un *challenge* ya no es una práctica segura! Cualquiera con acceso a la red puede observar los paquetes y obtener la contraseña en texto plano. Conociendo la contraseña tienes acceso al sistema.

Es por eso que se han diseñado mejores sistemas tipo *challenge-response* en el que la contraseña ya no se usa como respuesta directa del *challenge*, sino como herramienta para generar una respuesta derivada mediante cálculos matemáticos, aunque igual de válida. En términos generales podemos decir que el objetivo es demostrar que el usuario conoce la contraseña sin necesidad de enviarla tal cual al aire libre.





### 3. Técnicas de seguridad y autenticación

Conociendo el hecho de que no es seguro enviar una contraseña en texto plano para autenticarse, diversas variantes del challenge-response se pueden idear para resolver el problema de autenticar usuarios.

Algunos mecanismos importantes que se han ideado:

- Almacenar un texto derivado de la contraseña (por ejemplo un hash). No elimina del todo el problema pues si un atacante conoce la contraseña derivada puede acceder al sistema igual que con la contraseña original. La importancia de esta técnica radica en la complejidad de averiguar la contraseña original. Aunque un atacante se apropie de la contraseña derivada, al menos se mantiene cierto grado de privacidad con respecto a la original.
- Usar la contraseña para derivar una clave criptográfica que se usará para cifrar los mensajes entre el usuario y la plataforma. Esta técnica es muy importante pues elimina por completo la necesidad de enviar la contraseña, o cualquier texto derivado de esta, usándose únicamente para transformar los mensajes a enviar o recibir del servidor. Esta técnica es ampliamente usada. Tanto cliente como servidor conocen la contraseña y por ende pueden cifrar y descifrar mensajes entre sí.
- Uso de *tokens* en lugar de contraseñas. Un *token* es un conjunto de datos generados por el servidor y enviado al usuario a través de una vía segura. El usuario usará el *token* para autenticarse de igual manera que haría con una contraseña (ya sea enviando el *token* en sí o derivando una clave criptográfica de este). Los *tokens* tienen la ventaja de ser temporales (son caducos y volátiles), por lo que un *token* robado no será de mucho uso para un atacante en sesiones futuras. Suelen ser también generados a partir de información adicional que controla el servidor (timestamp, contador de uso, etc.), con lo que el servidor tiene mucho más conocimiento sobre el proceso de autenticación.



#### 4. Tecnologías actuales

En criptografía moderna hay dos soluciones aceptadas al cifrado y descifrado de mensajes, dependiendo de la naturaleza de la clave; simétrica y asimétrica.

En criptografía simétrica se usa la misma clave para cifrar y descifrar. En contraste, cuando usamos criptografía asimétrica se usa una clave diferente para cifrar y descifrar.

La tabla 4.1 nos muestra las diferentes tecnologías y nomenclaturas de cada sistema, comparadas lado a lado.

|                                      | <b>Cifrado Asimétrico</b>             | <b>Cifrado Simétrico</b>                 |
|--------------------------------------|---------------------------------------|--|
| <b>Sistema Central</b>               | Certificate Authority (CA).           | Authentication Server (AS).              |
| <b>Credenciales</b>                  | Certificate.                          | Ticket.                                  |
| <b>Clave</b>                         | Una para cifrar, otra para descifrar. | Una misma clave para cifrar y descifrar. |
| <b>Algoritmo de cifrado estándar</b> | RSA                                   | AES                                      |
| <b>Tamaño de claves (en bits)</b>    | 512, 1024, 2048                       | 128, 192, 256                            |

Tabla 4.1

## 4.1 Cifrado Asimétrico

En criptografía de clave asimétrica dos entidades intercambian su identidad así como claves públicas mediante lo que se conoce como *certificados*, los cuales son generados y firmados digitalmente por una *Certificate Authority* (CA, *Autoridad Certificadora*), la cual actúa en éste contexto como *Key Distribution Center* (KDC, *Centro de Distribución de Claves*).

Este método de autenticación y cifrado punto a punto forma la base de lo que se conoce como *Transport Layer Security* (TLS) o *Secure Socket Layer* (SSL), una capa extra que se añade a la pila del *Internet Protocol* (IP).

De manera (muy) simplificada, se pueden definir las siguientes acciones para realizar la autenticación de usuarios y el establecimiento de una comunicación segura:

- Ambas entidades reciben su certificado de la CA, el cual contiene la información que necesitan para identificarse y cifrar mensajes. Este paso lo podemos definir como la *distribución de credenciales*.

- Ambas entidades intercambian sus certificados, con el que se identifican y reciben la información necesaria para cifrar y descifrar correctamente los mensajes entre sí. Este paso en sí se puede concebir como la *autenticación de usuarios*.

La criptografía asimétrica tiene una serie de desventajas que hay que tomar en cuenta:

- El cifrado y descifrado asimétrico es muy lento en comparación con el cifrado simétrico. En un entorno donde afluayan muchos datos esto puede suponer una penalización en el rendimiento.

- La dependencia sobre la CA hace que muchos desarrolladores rechacen el uso de cifrado asimétrico o que prefieran el uso de certificados autofirmados, los cuales son más susceptibles a robo o falsificación.

- El uso de TLS/SSL no es una solución onnipotente ni la única solución que se debe adoptar. Sin embargo muchos desarrolladores dan por concluido el problema de la seguridad una vez se toma esta medida. El uso de certificados no debe ser excluyente de otras herramientas.

- El tamaño de las claves asimétricas es considerablemente más grande. Una clave asimétrica de 15360 bits tiene la misma seguridad que una clave simétrica de 256 bits.

## 4.2. Cifrado Simétrico

En contraste con el cifrado asimétrico tenemos la opción de cifrado simétrico, el cual resuelve el problema de velocidad y elimina la necesidad de certificados y una CA. Sin embargo seguimos necesitando una medida para autenticar usuarios que reemplace los certificados y un sistema central que haga el papel de la CA.

A este sistema central lo llamamos *Authentication Server* (AS, *Servidor de Autenticación*), y al conjunto de datos que determinan la identidad y credenciales de un usuario lo llamamos *Ticket*.

Aunque hay similitudes entre ambos sistemas, las diferencias son substanciales:

- Un servidor de autenticación (AS) puede ser implementado y mantenido por el propio desarrollador de la aplicación. En contraste, las CA's son organismos independientes que no pueden ser regulados ni modificados por los propios usuarios de este mecanismo.
- Un certificado suele tener una validez física indeterminada que puede ir de minutos a horas, días, meses o incluso años. En contraste los tickets sólo suelen tener validez para la sesión en la que se establecen; son mucho más volátiles.
- El establecimiento de credenciales es más directo en el cifrado asimétrico, necesitando solo del intercambio de certificados entre las entidades correspondientes. En el cifrado simétrico sin embargo siempre se necesita de la supervisión del servidor de autenticación para garantizar una comunicación libre de fallos e inconsistencias.
- La velocidad de cifrado y descifrado es mayor en los algoritmos simétricos.
- El tamaño de las claves simétricas es considerablemente menor que las asimétricas, sin perder efectividad. Una clave simétrica de 128 bits es equivalente en seguridad a una clave asimétrica de 1024 bits.

El servidor de autenticación de usuarios más ampliamente conocido y estudiado es *Kerberos*, y es la base que forma el sistema que estudiaremos en este documento.

### 4.3. Kerberos

Para resolver el problema de autenticar usuarios, el *Massachusetts Institute of Technology* (MIT, *Instituto Tecnológico de Massachusetts*) diseñó e implementó un protocolo que permite a dos entidades intercambiar credenciales y establecer una comunicación de manera segura. Resolviendo los dos problemas principales en la comunicación de datos:

- 1- Autenticación de usuarios: identificar correctamente las entidades participantes sin dar lugar a duda sobre posibles impostores.
- 2- Distribución de claves: permitir ambas entidades establecer una comunicación segura lejos de la escucha de terceros.

El protocolo Kerberos hace uso de *tickets* para proporcionar identidad a las entidades, similar a los certificados en el cifrado asimétrico.

Hay varios tipos de *tickets* y su uso depende del instante en el tiempo en el que ejecutamos el protocolo. Dicho de otra forma: el *ticket*, así como sus datos y significado, es cambiante y caduco; tiene alta volatilidad.

Podemos resumir el funcionamiento de Kerberos en los siguientes pasos:

1- Suponemos Alice como el usuario de una plataforma web y Bob como la plataforma en sí (el proveedor de servicios)<sup>1</sup>. Suponemos C<sup>2</sup> como el servidor de autenticación Kerberos, compuestos por el *Authentication Server (AS)* y el *Ticket Granting Server (TGS)*.

2- El usuario Alice realiza un *handshake* con el servidor AS, el cual proporciona a Alice un *ticket* especial llamado *Ticket Granting Ticket*.

En Kerberos éste *ticket* contiene:

- Información cifrada con una clave maestra que solo conoce C.
- Un número aleatorio.
- Demás información necesaria para el protocolo y el usuario.

El TGT está completamente cifrado usando la contraseña de Alice.

3- El usuario Alice debe descifrar el TGT, incrementar el número aleatorio en uno y volver a cifrarlo. El nuevo TGT se devuelve al TGS.

1 En ciberseguridad, cuando se explica un protocolo de comunicación entre dos entidades, es común usar los nombres Alice y Bob para referirse a las entidades comunicantes.

2 El nombre de entidades terceras suele variar y depender del contexto. Para nuestro caso, usamos C como referencia al servidor de autenticación y David para referirnos a atacantes.

4- Una vez llega el TGT modificado por Alice al TGS, se realizan las comprobaciones pertinentes y se genera un *Service Granting Ticket* (SGT), que el usuario Alice puede usar para autenticarse con Bob. El SGT contiene la clave de sesión para comunicar el usuario y el proveedor.

En la figura 4.1 se muestra el proceso del protocolo.

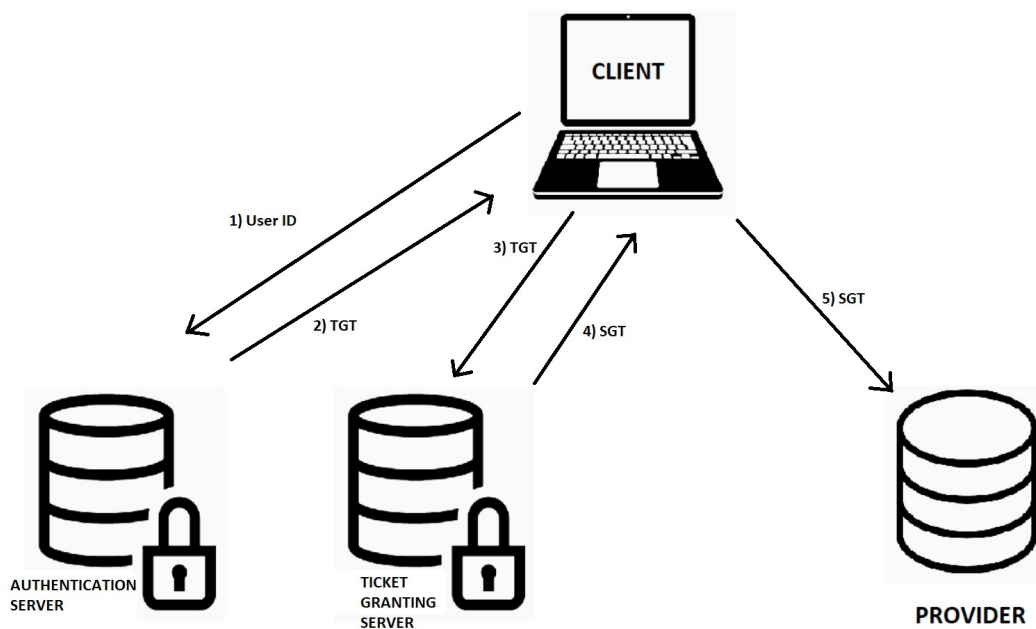


Figura 4.1

Resumiendo:

- 1) El usuario envía su ID al AS de Kerberos para iniciar la autenticación.
- 2) Kerberos responde con el *Ticket Granting Ticket* (TGT).
- 3) El usuario modifica el TGT y lo envía al TGS de Kerberos.
- 4) Kerberos responde con el *Service Granting Ticket*.
- 5) El usuario usa el SGT para autenticarse con el proveedor de servicios.

El protocolo de Kerberos ofrece una serie de ventajas sobre otras soluciones que lo hace ideal para ciertos escenarios, sin embargo Kerberos brilla por su ausencia en Internet y esto es debido a una serie de limitaciones que tiene:

- El protocolo fue diseñado con antiguas tecnologías (DES, 3DES, RC4, MD4, etc) que actualmente se consideran extremadamente inseguras. Ciertamente es que las más novedosas versiones implementan las tecnologías más modernas (SHA2, AES e incluso cifrado asimétrico), sin embargo el uso de estas tecnologías es puramente opcional, muchas veces no siendo por defecto. Es incluso peor el hecho de que Kerberos permite a los usuarios decidir qué tecnologías usar (permitiendo a un atacante forzar el uso de algoritmos débiles). Incluso la contraseña del usuario es almacenada usando las distintas funciones *hash* conocidas, siendo posible para un atacante enfocarse en la que usa el *hash* más débil (con más posibilidad de colisiones).
- El protocolo en sí solo está escrito y definido para el lenguaje de programación C y derivados. No hay implementaciones en otros lenguajes. Intercomunicar una aplicación cliente escrita en Java con un servidor escrito en Python mediante Kerberos es simplemente imposible sin tener que hacer uso de wrappers o las primitivas de los lenguajes para acceder a código C (Java Native Interface, Python Ctypes, etc.). Mención especial merece el hecho de que el protocolo hace uso de datos en binarios para la comunicación, omitiendo el uso de estándares tales como XML o JSON para el intercambio de datos.
- El protocolo es complejo, no solo en el número de entidades involucradas, sino también en el intercambio de mensajes y el significado de estos. Demasiados paquetes van y vienen de muchos sitios y eso aumenta la probabilidad de fallo en la red o de escucha por parte de entidades no deseables. Pese a esto, el proveedor de servicios nunca realiza una verificación del SGT recibido por el usuario, permitiendo ataques de tipo *Golden Ticket* (detallado más adelante).
- Hay un motivo por el que Kerberos no está en uso actualmente por Internet: la forma en la que el servidor evita ataques de tipo *replay* (lo veremos más adelante) es mediante el uso de *timestamps* (marcas de tiempo) que deben actualizarse con cada intercambio de datos. Esto requiere que todas las entidades involucradas tengan su reloj interno extremadamente sincronizado, lo cual nunca es el caso cuando hablamos de computadoras en extremos diferentes del planeta (Internet). Un pequeño fallo eléctrico puede hacer que el reloj deje de estar sincronizado, un retraso en la red o un incluso un paquete perdido que se debe retransmitir puede dejar el ticket en un estado inservible. Simplemente estamos ante una situación en la que necesitamos unas condiciones teóricas idóneas que son irrealizables en la práctica.



Un servidor dedicado para la autenticación de usuarios es una buena aproximación para reutilizar código. Siempre es mejor poder incorporar un sistema ya creado para identificar a los usuarios en lugar de reescribir el proceso de autenticación desde cero.

Hay demasiadas técnicas que los atacantes pueden usar para romper la seguridad de la autenticación y tener que tomar todas en cuenta cada vez que se crea una nueva plataforma web es un trabajo tedioso. Teniendo el tiempo y demás recursos en contra es muy fácil obviar ciertos detalles. Todo esto deriva en que cada página de autenticación tiene su propia forma de verificar la identidad del usuario, algunas más vulnerables que otras, algunas resistentes a ciertos ataques pero no a otros, etc.

Un ejemplo práctico conocido de dudosa seguridad es el uso de OAuth2 para autenticar usuarios. Muchas páginas web usan esta tecnología para sus páginas de *login*, obviando el hecho de que OAuth2 no es una herramienta de autenticación, sino de autorización.

Recordemos la diferencia entre autenticación y autorización:

- Autenticación es la capacidad de un sistema de determinar la identidad de un usuario, es decir, poder afirmar que el usuario es quien dice ser.
- Autorización es la capacidad de un sistema de permitir acceso a cierto recurso dependiendo del usuario que accede. La autorización evita que un usuario acceda a un recurso sin los permisos pertinentes, pero no identifica que el usuario accediendo es realmente quien dice ser.

En otras palabras, OAuth2 sólo protege el acceso a usuarios de ciertos recursos (autorización), pero no verifica la identidad del usuario que accede al recurso (autenticación).

Es fácil entender por qué muchas webs usan OAuth2 para el *login* ya que es importante tener un mecanismo que se encargue del proceso de autenticación de manera automatizada y reutilizable; que el desarrollador pueda hacer uso de unas librerías ya escritas y conocidas. Sin embargo este no es el objetivo de OAuth2, y Kerberos es una buena base, pero sus defectos imposibilitan su uso en entornos cambiantes y donde hay poca o ninguna sincronía entre los sistemas comunicantes, así como la complejidad de su API y falta de transparencia en los procesos criptográficos. Es por ello que en este proyecto se propone una alternativa basada en Kerberos que intenta solventar los problemas inherentes en su funcionalidad.



## **5. Ingeniería de Software: diseño y modelado**

Antes de proceder a la implementación e implantación de nuestro sistema software, es imprescindible definir los requisitos, las arquitecturas a usar y el diseño en general de la aplicación. Para ello estudiaremos la funcionalidad del proyecto desde un punto de vista de ingeniería de software.

En este apartado estudiaremos lo siguiente:

- Los requisitos del sistema, su descripción, dependencias y objetivo.
- Los diferentes casos de uso que debe contemplar el sistema software.
- Los diferentes diagramas que expliquen el funcionamiento de los componentes software.
- La planificación necesaria para llevar a cabo el proyecto.
- La arquitectura necesaria para construir el sistema.

### 5.1. Tabla de requisitos

Los requisitos del sistema software nos permiten realizar un estudio a priori del funcionamiento del sistema software en relación a los objetivos a cumplir. Para ello se procede a enumerar los diferentes requisitos y expandir su descripción así como la relación con otros requisitos.

| Nombre | Tipo        | Descripción  | Antecedentes           | Sucesores             |
|--------|-------------|--|------------------------|-----------------------|
| RE001  | Funcional   | El sistema debe identificar y autenticar una comunicación entrante con usuario y contraseña.   | -                      | RE002, RE003<br>RE005 |
| RE002  | Estructural | El sistema debe ocultar la contraseña del usuario en la Base de Datos.   | RE001,<br>RE008        |                       |
| RE003  | Funcional   | El sistema debe omitir la contraseña del usuario en la transmisión de datos.   | RE001,<br>RE008        | RE004                 |
| RE004  | Funcional   | El sistema debe permitir a un usuario identificarse con credenciales temporales.   | RE003                  | RE007                 |
| RE005  | Funcional   | El sistema debe permitir a un usuario autenticar las credenciales de otro usuario.   | RE001                  | RE007                 |
| RE006  | Funcional   | Los usuarios del sistema deben poder identificar al sistema.   |                        |                       |
| RE007  | Funcional   | El sistema debe facilitar a dos usuarios intercambiar claves de cifrado de manera segura.  | RE004,<br>RE005        |                       |
| RE008  | Sistema     | El sistema debe prevenir ataques por fuerza bruta.   |                        | RE002, RE003          |
| RE009  | Sistema     | El sistema debe prevenir escuchas e interferencias de entidades ajenas a la comunicación.  |                        |                       |
| RE010  | Sistema     | El sistema debe ser agnóstico con el entorno a ejecutar.   |                        | RE011<br>RE013, RE014 |
| RE011  | Usuario     | Las librerías del cliente deben estar presente en los principales lenguajes de programación y garantizar la compatibilidad entre sistemas software y hardware. | RE010                  |                       |
| RE012  | Sistema     | Un fallo en la seguridad no debe comprometer las credenciales originales del usuario.  | RE002,<br>RE003, RE008 |                       |
| RE013  | Estructural | El intercambio de datos entre los actores debe estar estandarizado mediante formatos comunes.  | RE010                  |                       |
| RE014  | Sistema     | El sistema debe hacer uso de funciones y librerías criptográficas estándar y ampliamente conocidas.  | RE010                  |                       |

Tabla 5.1

## 5.2. Tabla de tareas

Tareas a realizar para la implementación del proyecto.

| Nombre  | Descripción  | Requisitos                        | Predecesora  |
|---------|--|-----------------------------------|--------------|
| TA001   | Crear tablas de la BBDD usando Django.   | RE001, RE002                      |              |
| TA002   | Crear API REST para autenticar usuario usando Django.  | RE001, RE013                      | TA001        |
| TA003   | Crear interfaz criptográfica en Python y JavaScript haciendo uso de cifrados estándares y claves de tamaño considerable.   | RE002, RE003, RE010, RE011, RE008 |              |
| TA004   | Crear librería del cliente en JavaScript y Python para facilitar el uso del protocolo en aplicaciones web.   | RE010, RE011, RE013               | TA003        |
| TA005   | Crear página web de prueba del login.  | RE001                             | TA002, TA004 |
| TA006   | Modificar login para omitir la contraseña usando otro método de autenticación basado en tokens.  | RE001, RE003                      | TA005        |
| TA007   | Proteger proceso de autenticación contra ataques tipo Replay.  | RE009                             | TA006        |
| TA008   | Proteger la contraseña del usuario contra ataques de fuerza bruta mediante el uso de una <i>Key Derivation Function</i> estándar y mensajes de error genéricos.                                | RE008, RE012                      | TA006        |
| TA009   | Permitir a un usuario ya autenticado cambiar sus credenciales permanentes (contraseña) sin la posibilidad de escucha por terceros.   | RE003, RE008                      | TA006        |
| TA010   | Permitir a un usuario establecer una clave de sesión con otro usuario de manera segura.  | RE004, RE005, RE007               | TA006        |
| TA010.1 | Permitir a un usuario verificar las credenciales de otro usuario.  | RE005                             | TA006        |
| TA010.2 | El servidor debe generar una clave de cifrado común para dos usuarios que establecen una sesión. La clave debe estar cifrada para estos dos usuarios.  | RE007                             | TA010.1      |
| TA010.3 | La librería del cliente debe facilitar el uso de las claves de sesión para cifrar y descifrar datos.   | RE007                             | TA010.2      |
| TA011   | Crear ejemplo de proveedor de servicios que requiera el uso de credenciales temporales y claves de sesión para obtener datos confidenciales.   | RE007                             | TA010        |
| TA012   | Crear versiones de la interfaz criptográfica y la librería del cliente para los lenguajes de programación Java, C y C++ con la misma funcionalidad que las versiones para JavaScript y Python. | RE010, RE011                      | TA010        |

Tabla 5.2

5.3. Diagrama de Gantt

El siguiente diagrama muestra, de manera visual, las tareas a realizar en el proyecto y el tiempo de ejecución de cada tarea así como la concordancia con el resto de tareas.

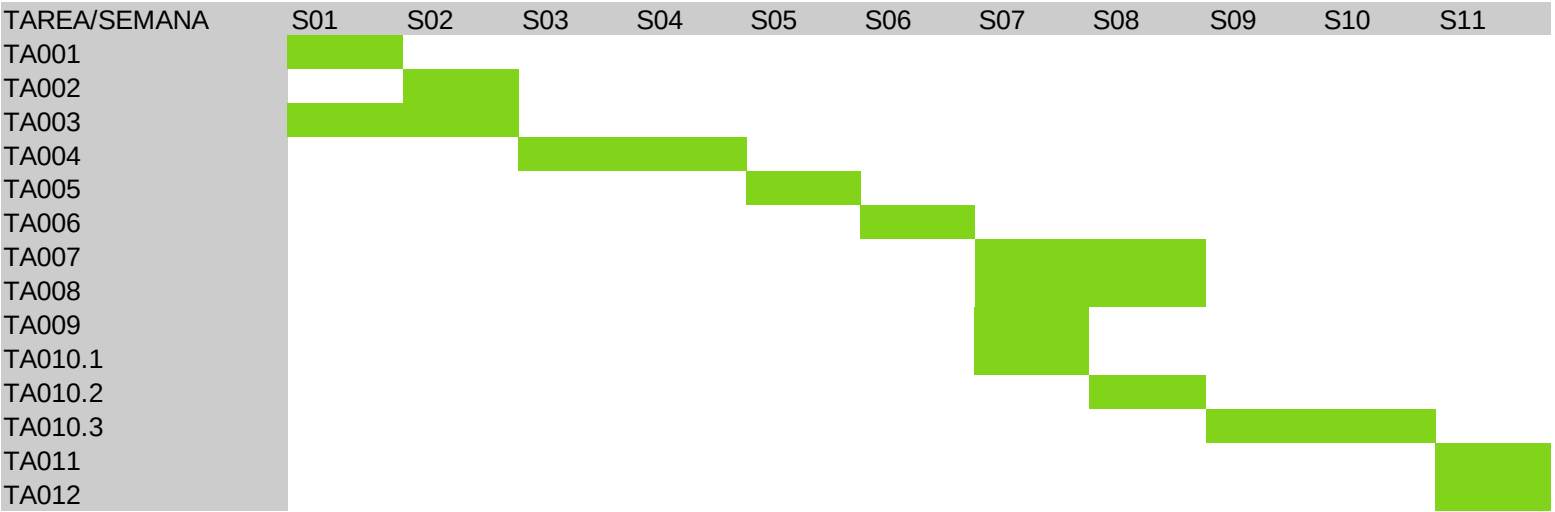


Tabla 5.3

#### 5.4. Diagrama de Casos de Usos

Para comprobar el funcionamiento del sistema es importante primero estudiar la utilidad del sistema desde la perspectiva del usuario y las operaciones que puede realizar.

Para ello podemos hacer uso del diagrama de casos de uso (figura 5.1) a continuación.

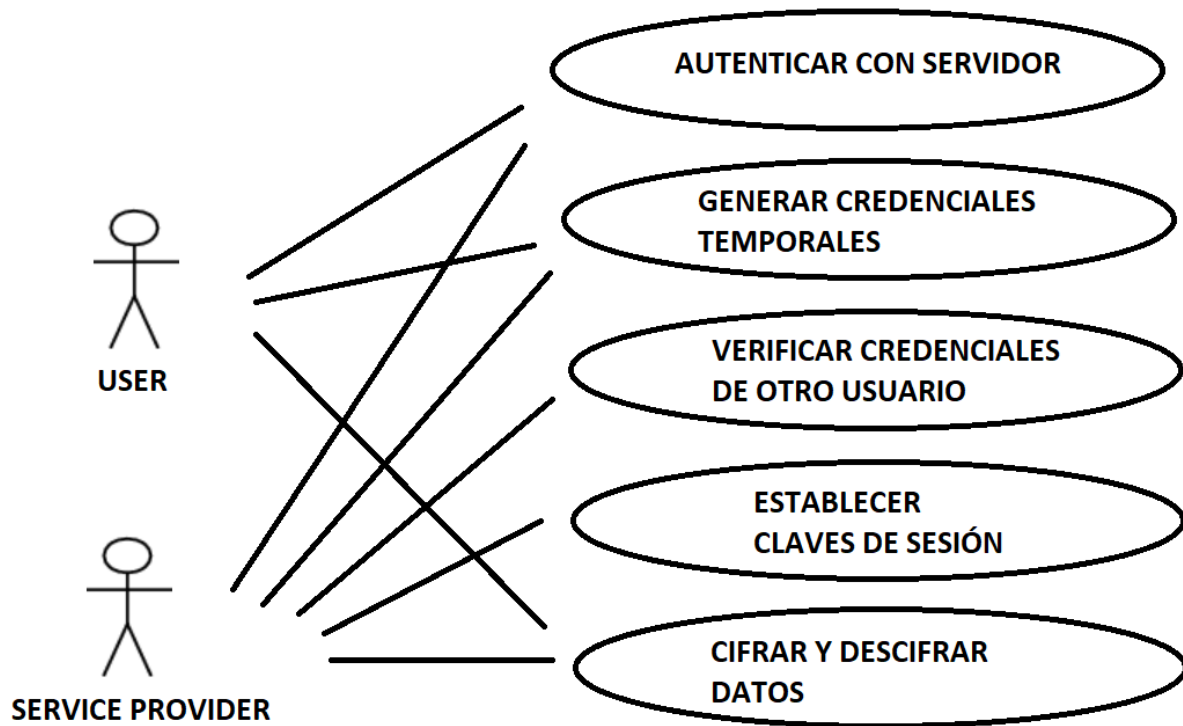


Figura 5.1

### 5.5. Diagrama del protocolo

El diagrama de protocolo nos permite observar el intercambio de datos entre las entidades para estudiarlo en concordancia con el análisis de red del sistema (que veremos en el apartado de implementación).

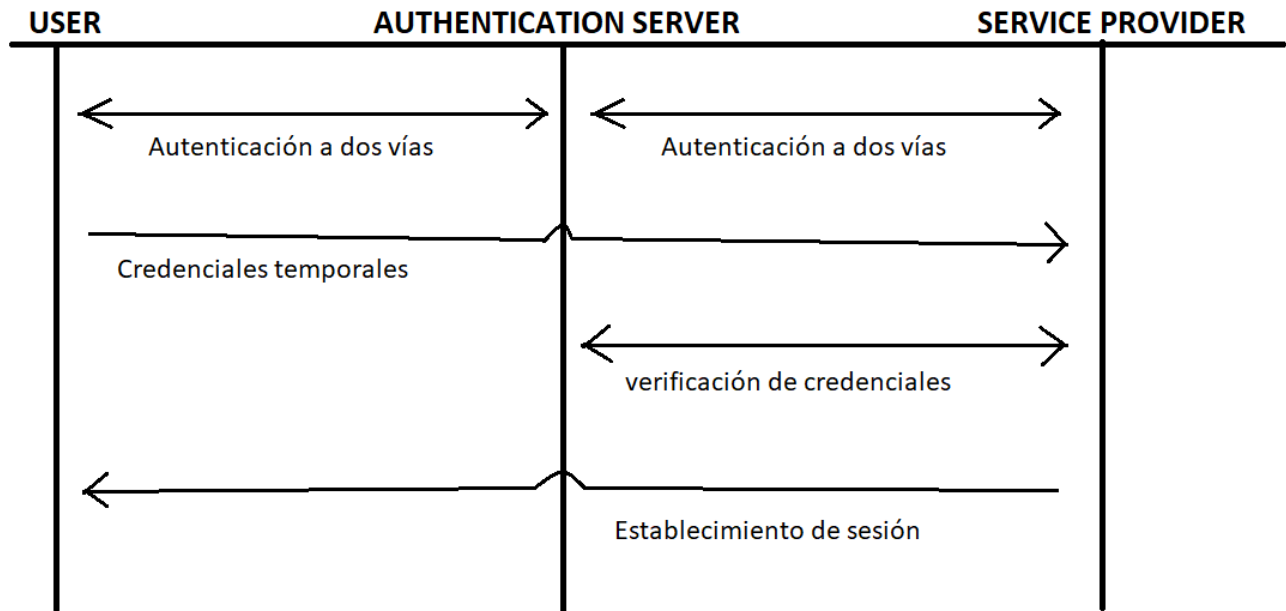


Figura 5.2



## 5.6. Diagrama de flujo

A su vez, es importante estudiar el flujo de ejecución del sistema en general para tener una idea clara del funcionamiento del protocolo y la actuación de los diferentes actores ante la llegada de datos.

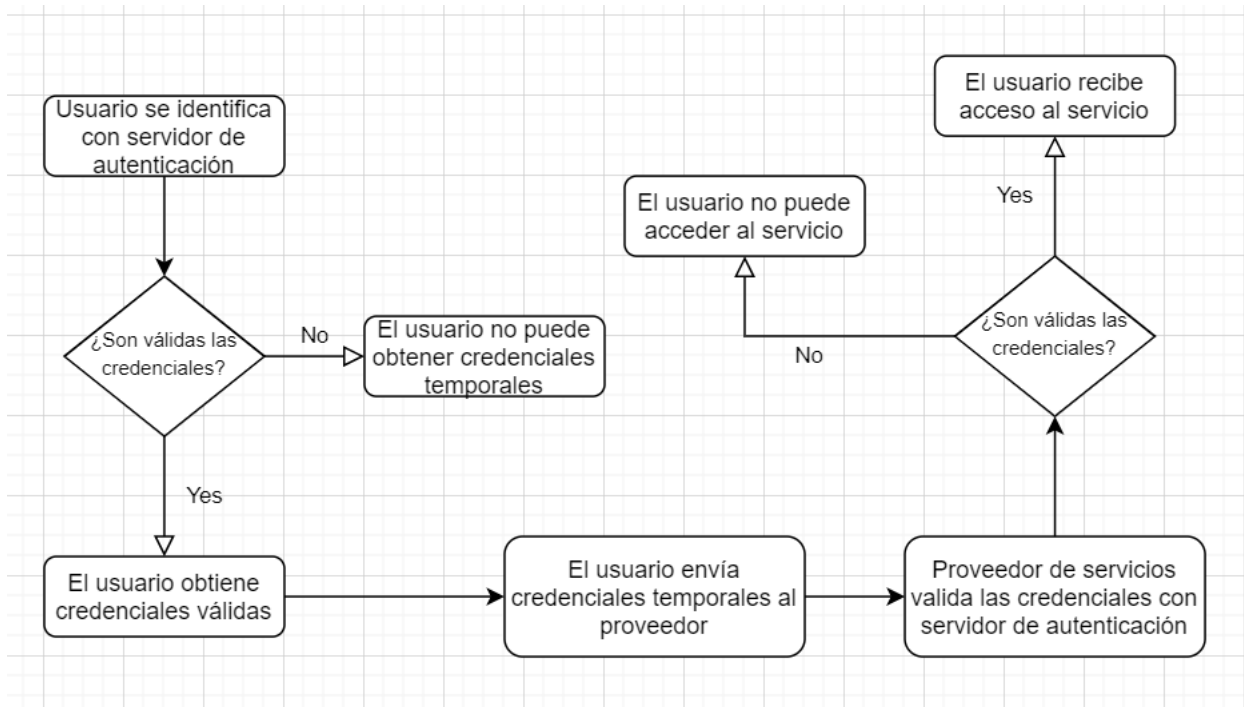


Figura 5.3

### 5.7. Diagrama Entidad-Relación de la Base de Datos

La base de datos del sistema debe permitirnos almacenar usuarios, las credenciales temporales que se van generando por cada autenticación (los *tickets*) y las sesiones que se establecen entre distintos usuarios, siguiendo las siguientes especificaciones.

- Un usuario puede generar muchas credenciales temporales, una por cada autenticación, pero sólo una es la credencial actualmente válida. Una credencial es generada para un sólo usuario.
- La sesión se establece entre dos usuarios usando credenciales temporales. La sesión debe asociarse a las credenciales que se usaron en el momento de ser establecida dicha sesión.

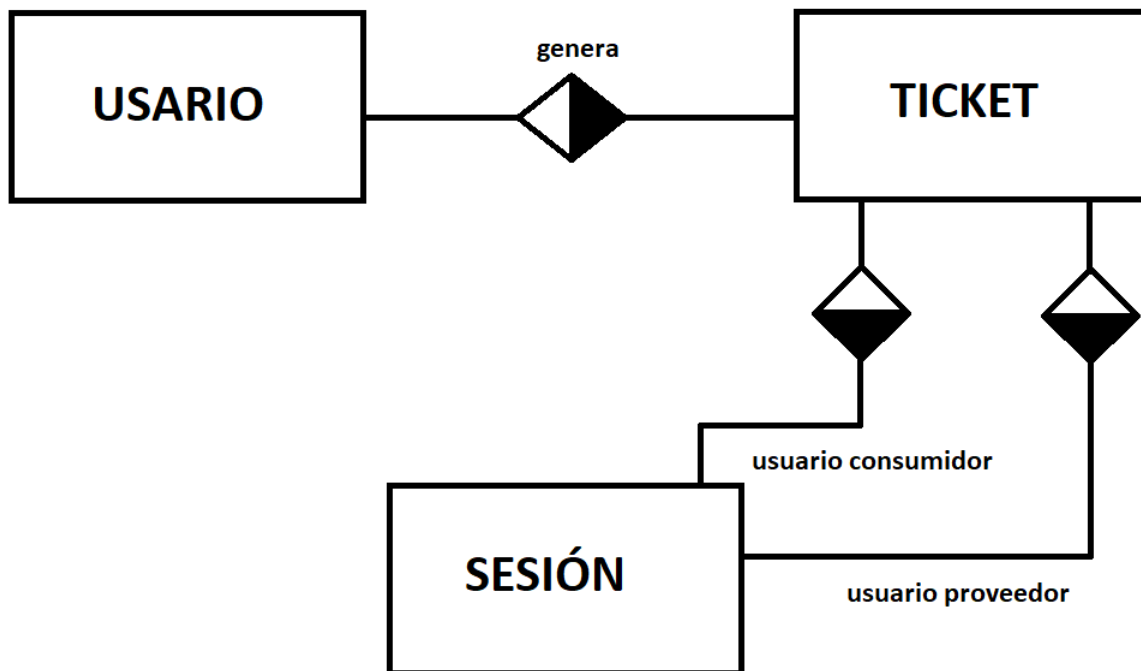


Figura 5.4

## 6. Protocolo Caronte: Introducción

El protocolo Caronte surge como una reimaginación y modernización del protocolo Kerberos. Siendo muy similares en su apariencia externa y técnicas criptográficas pero con substanciales cambios internos para resolver los problemas a los que se enfrenta Kerberos en la actualidad.

Principalmente los problemas a resolver (y las soluciones propuestas por Caronte), así como diferencias en el protocolo, son las siguientes:

- Uso de primitivas criptográficas diversas sin establecer un estándar fijo. El principal problema al que se enfrenta Kerberos en este aspecto es la necesidad de mantener compatibilidad con software tipo *legacy*. En sus inicios Kerberos usaba diferentes funciones *hash* y de cifrado que actualmente ya están en desuso. El problema radica en la falta de transparencia en las funciones criptográficas; el cliente o usuario del servidor de autenticación no debería tener conocimiento directo de las funciones usadas. En contraste, usando una interfaz que proporciona de manera genérica la funcionalidad requerida por el cliente sin exponer directamente la funcionalidad criptográfica subyacente podría permitir en futuras versiones cambiar por completo las funciones y técnicas usadas sin suponer gran cambios en la funcionalidad o uso de la librería por parte del cliente. Efectivamente eliminando la dependencia con cualquier función que pueda ser considerada vulnerable en un tiempo futuro.
- Falta de compatibilidad con diversos sistemas. Kerberos está escrito en C, lo cual es una gran ventaja a la hora de portabilidad. Sin embargo no hay apenas librerías propias para otros lenguajes más allá del uso de wrappers. La representación interna de los datos no es estándar y requiere de software específico para interpretarla. Esto dificulta la intercomunicación entre sistemas con diferente software y hardware. La solución es bastante trivial; hacer uso de estándares de transmisión de datos en Internet tales como XML o JSON. Debido a su gran disponibilidad y potencial para crear APIs REST, el formato usado por Caronte para transmitir datos es JSON, el cual tiene soporte nativo en Java, Python, JavaScript y demás lenguajes, así como una variedad de librerías disponibles en C (CJSON, Jansson y demás).
- El uso de *timestamps* para autenticar mensajes y evitar ataques tipo *replay*, lo cual imposibilita que dos computadoras con diferentes relojes puedan comunicarse de manera segura. La solución a este problema radica en cambiar por completo el uso de *timestamp* por otro mecanismo que nos permita defendernos contra ataques *replay*. Sea cual sea, este mecanismo tiene que cumplir las siguientes dos condiciones: (1) tanto cliente como servidor deben estar 100% sincronizados con los datos a verificar (ambos conocen su valor actual), y (2) un cambio en los datos debe hacer que todos los mensajes enviados

con valores previos queden invalidados (por ende un ataque *replay* es imposible). Caronte reemplaza los *timestamps* por un contador que realiza la misma funcionalidad (lo detallaremos más adelante).

- El protocolo Kerberos requiere de dos servidores principales; el *Ticket Granting Server* y el *Key Distribution Center*. Esta separación genera confusión sobre qué servidor tiene qué responsabilidad. Para simplificar las cosas, Caronte hace uso de un solo servidor que actúa de TGS y de KDC a la vez, dependiendo de la funcionalidad que requiera el cliente y según la fase de autenticación en la que se encuentre.

- Caronte no hace ningún tipo de distinción entre clientes y servidores; ambos son usuarios del servicio y ambos deben autenticarse con Caronte previo al establecimiento de su comunicación. Esto permite a las entidades comunicantes establecer su propio método de comunicación (ya sea cliente-servidor o p2p). Esto también resulta en el hecho de que Caronte no controla los privilegios de acceso del usuario.

- En Kerberos, el cliente recibe el TGT, lo modifica y lo envía al servidor para recibir el SGT como respuesta. En Caronte, el SGT es el resultado de modificar el TGT por parte del usuario. En otras palabras: en Kerberos es el servidor quien crea los SGT, mientras que en Caronte los genera el propio usuario.

- Kerberos proporciona claves de sesión para permitir una comunicación segura entre las entidades autenticadas. Sin embargo, el uso de estas claves no es estándar ni está reforzado y las primitivas criptográficas que permiten hacer uso de estas claves no están pensadas para las aplicaciones web modernas. Esto implica que Kerberos se suele usar únicamente en el proceso de autenticación del usuario, pero no en el cifrado de la comunicación entre usuarios. Caronte por el contrario refuerza el uso de estas claves mediante el KGT y proporciona funcionalidad para cifrado de datos.

- En Kerberos, el proveedor de servicios no valida el SGT del usuario con el AS. En Caronte esta validación es obligatoria para establecer la clave de sesión.

- Los *tickets* de Kerberos contienen datos sobre el usuario y sus permisos y privilegios con respecto a la aplicación. En Caronte el aspecto de privilegios y accesos es delegado al proveedor de servicios y su contexto interno. Esto implica que en Caronte no existe un usuario administrador; todos los usuarios tienen exactamente el mismo acceso y privilegio a los servicios de Caronte y son todos igual de fiables (o no fiables).

Sabiendo esto, veamos ahora el funcionamiento del protocolo:

- 1- Suponemos un usuario de una aplicación al que llamaremos Alice.
- 2- Suponemos un proveedor de servicios web al que llamaremos Bob.
- 3- Suponemos un servidor de autenticación, en nuestro caso Caronte.
- 4- Suponemos un Man-In-The-Middle al que llamaremos David, que escucha y posee todos los paquetes enviados entre Alice, Bob y Caronte.

El objetivo a conseguir es el siguiente:

A- Autenticar usuarios: Alice, Bob y Caronte se identifican como tal.

B- Distribuir claves: Mediante Caronte, Alice y Bob obtienen una clave secreta que sólo ellos conocen, la cual será usada para cifrar su comunicación.

Veamos primero el proceso de autenticación entre un usuario y Caronte.

En este caso, el proceso es idéntico tanto para Alice como para Bob:

I- Alice/Bob envían su identificador. Generalmente se suele usar el email o un texto derivado de este. Caronte refuerza el uso de un texto derivado del email en lugar del email en sí, para evitar que se conozca la identidad real del usuario en cuestión.

II- Caronte usa el identificador para determinar el usuario y su contraseña preestablecida. En este paso Caronte genera el *Ticket Granting Ticket*, similar a Kerberos, y se lo envía de vuelta al usuario. El TGT contiene, entre otras cosas, un token que el usuario usará de ahora en adelante como reemplazo de su contraseña, todo ello cifrado usando una clave derivada de la contraseña original del usuario.

III- El usuario obtiene el TGT, lo descifra y usa la información que contiene para generar el *Service Granting Ticket*. A estas alturas el usuario ya tiene la información que necesita para autenticarse, sin hacer uso de su propia contraseña (sólo se usa para descifrar el TGT y transformarlo en el SGT).

IV- El usuario devuelve el SGT a Caronte para confirmar que ha sido capaz de descifrar el TGT. Es en este punto, cuando el *ticket* del usuario queda validado por parte de Caronte. El usuario no puede usar el SGT hasta que no lo ha validado con Caronte. Este paso es esencial, pues el SGT contiene información para prevenir ataques tipo *replay*, e información que debe ser inicializada correctamente por el usuario (Caronte valida este dato cada vez que recibe un SGT). Más adelante veremos cómo funciona internamente el SGT para evitar ataques tipo *replay*.

V- En este punto David tiene los *tickets* tanto de Alice como de Bob, aunque no tiene sus claves para descifrarlos, sí que puede emplear otras técnicas como veremos más adelante.

Es en este punto que el usuario (tanto Alice como Bob) tiene su *Service Granting Ticket* validado por Caronte. El SGT será usado por el usuario para autenticarse con otro usuario.

Este proceso es un poco ambiguo, ya que Caronte reconoce a ambos como usuarios sin hacer distinción; no podemos saber con certeza quién será cliente y quién servidor (o incluso si es una comunicación *Peer-To-Peer*). Aunque en el mundo real es Alice (el usuario) quien se autentica con el servidor (Bob), la única diferencia real aquí radica en quien le envía su SGT a quien.

Tanto Alice como Bob tienen un SGT propio de cada uno. Sería fácil pensar que Alice y Bob intercambien sus SGT para la autenticación, pero hay que tener en cuenta:

- Alice y Bob ya se han autenticado con Caronte, no tiene sentido que vuelvan a autenticarse el uno con el otro.
- Para verificar un *ticket* hay que enviarlo a Caronte. Si verificamos ambos *tickets* por ambas partes hablamos de una gran cantidad de mensajes entre Alice, Bob y Caronte.
- Caronte conoce las claves de Alice y Bob, pero Alice no conoce la de Bob y viceversa tampoco: no serían capaces de cifrar/descifrar mensajes el uno con el otro. Se necesita una nueva clave compartida por ambos y proporcionada de manera segura.

La solución es más simple; solo se necesita que un usuario valide los *tickets* (tanto el suyo propio como el del otro usuario) para que Caronte genere una clave compartida.

Es aquí cuando entra en juego lo que llamaremos el *Key Granting Ticket* (KGT), el cual es un *ticket* especial que contiene la información y autenticación de ambos usuarios, tanto Alice como Bob. En otras palabras, un KGT no es más que un SGT modificado para contener el SGT de otro usuario que ha pedido establecer una comunicación. Es mediante el KGT que Caronte valida la petición de comunicación entre Alice y Bob y genera una respuesta en forma de clave criptográfica para ser usada entre Alice y Bob. Esta clave de sesión está cifrada para Alice y Bob, de manera que sólo estas entidades tendrán acceso a dicha clave.

El protocolo se resume en los siguientes pasos:

1- Alice envía su SGT a Bob. Suponiendo que el SGT de Alice ha sido verificado por Caronte (paso 4 de la autenticación con Caronte).

2- Bob incrusta el SGT de Alice dentro de una versión modificada de su propio SGT, convirtiéndolo así en un KGT. Suponiendo que el SGT de Bob ha sido verificado por Caronte (paso 4 de la autenticación con Caronte).

3- Bob envía el nuevo KGT a Caronte, el cual verifica la identidad de ambas partes involucradas. Caronte genera una nueva clave aleatoria que podrán usar Alice y Bob para cifrar su comunicación. Se crean dos copias de la misma clave, una cifrada usando la clave secreta de Alice y otra usando la clave secreta de Bob. Caronte devuelve ambas copias a Bob.

4- Bob recibe su copia de la clave de sesión, la descifra con su clave secreta y devuelve su copia a Alice.

5- Alice descifra la clave de sesión. En este punto tanto Alice como Bob comparten una clave secreta que solo ellos han podido descifrar (puesto que las únicas copias que han circulado en la red estaban cifradas para Alice y Bob).

David no conoce la contraseña de Alice ni la de Bob, con lo que no importa que tenga acceso a todos los paquetes en la comunicación, nunca será capaz de descifrar la clave de sesión para poder leer y escribir en el canal de comunicación entre Alice y Bob.

En este punto se ha establecido una sesión cifrada entre Alice y Bob usando una clave aleatoria y volátil.

Como se puede observar el protocolo Caronte en sí es un derivado de Kerberos.

Se mantiene gran parte de la nomenclatura y significado de las diversas entidades así como los objetivos. Cambiando principalmente qué mensajes se envían y dónde, así como el funcionamiento interno.

Podemos usar la figura 6.1 a modo ilustrativo y resumen del protocolo, paso a paso.

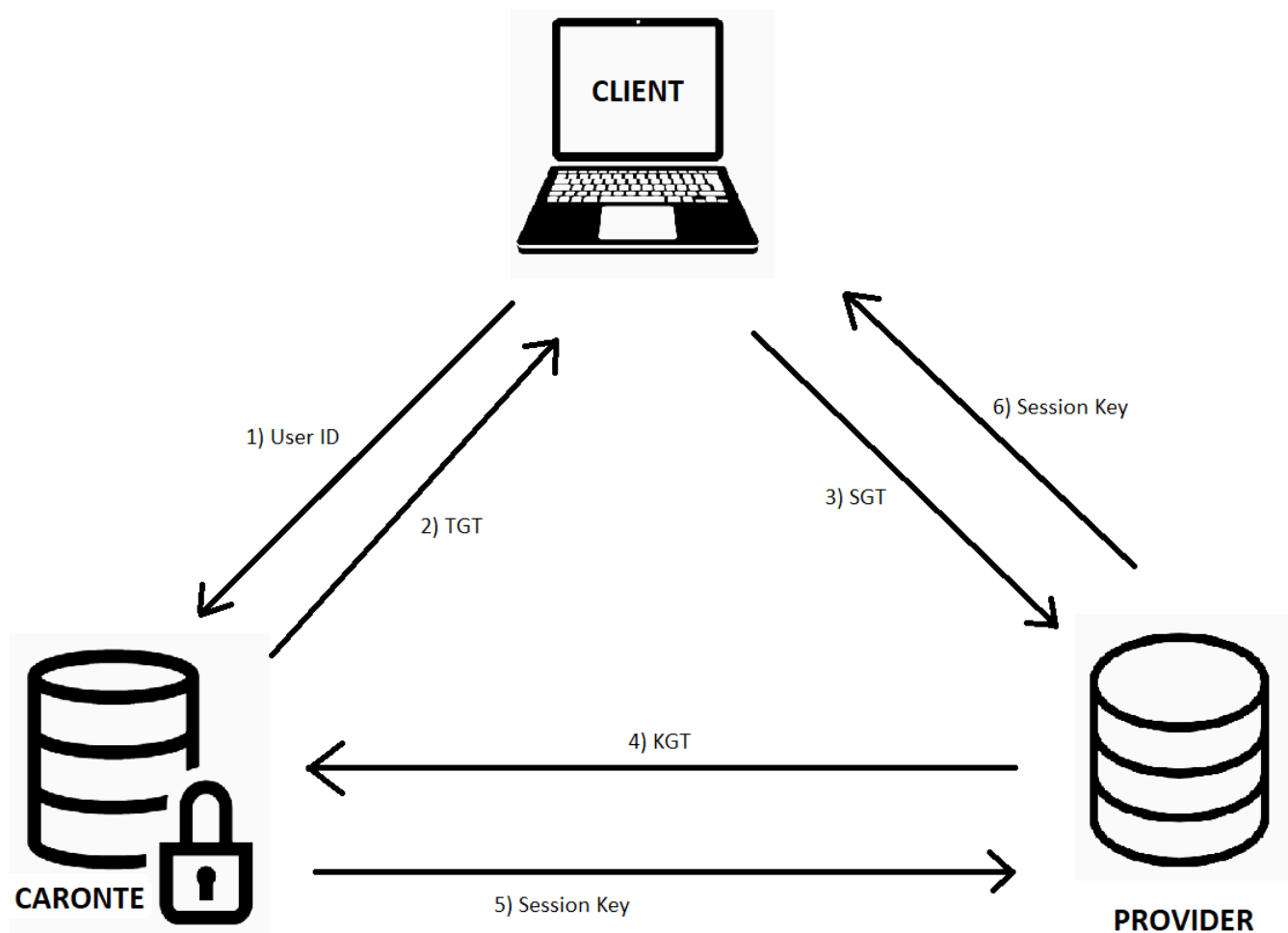


Figura 6.1

Cada paso tiene su significado en el protocolo:

- 1) El usuario envía su identificador para iniciar sesión con Caronte.
- 2) Caronte responde al usuario con el *Ticket Granting Ticket*.
- 3) El usuario genera el *Service Granting Ticket* y lo envía al proveedor.
- 4) El proveedor genera el *Key Granting Ticket* y lo envía a Caronte.
- 5) Caronte responde con las claves de sesión para el usuario y el proveedor.
- 6) El proveedor envía la copia de la clave de sesión para el usuario.



## 7. Principales ataques cibernéticos y contramedidas

Para poder entender el funcionamiento interno de Caronte (lo datos que hay en un ticket y para qué se usan) es imprescindible entender las principales vías y técnicas de ataque a las que nos enfrentaremos.

Es importante entender que no todos los sistemas van a ser vulnerables a los mismos ataques, y no todas las vulnerabilidades son igual de críticas. Es por eso que estudiaremos las diferentes técnicas existentes contrastándolas con la funcionalidad de Caronte para determinar cómo puede defenderse Caronte de estos ataques.

Veremos más detalladamente los diferentes ataques que encontramos en la tabla 7.1.

| Nombre                  | Descripción   | Importancia |
|-------------------------|---|-------------|
| Man-In-The-Middle       | Entidades ajenas al canal de comunicación con el objetivo de interceptar los mensajes.                            | Muy Alta    |
| Ataque Replay           | Reenvío de paquetes generados por las entidades comunicantes con la intención de falsificar la autenticación.     | Muy Alta    |
| Ataque por fuerza bruta | Uso de una herramienta automatizada para probar todas las posibles combinaciones hasta dar con la clave correcta. | Alta        |
| Ataque Selfie           | Uso de las propias credenciales del servidor para autenticarse como usuario.                                      | Baja        |
| Inyección de Código     | Uso de datos malformados con la intención de modificar el flujo de ejecución del servidor.                        | Media       |
| Ataques de Kerberos     | Uso de técnicas específicas que atacan el protocolo Kerberos.   | Muy Baja    |

Tabla 7.1

## 7.1. *Man-In-The-Middle*

Quizás el ataque más común y efectivo de los que se conoce. La premisa es simple: siempre hay alguien escuchando la transmisión de datos entre dos entidades. A este “alguien” se le conoce como *Man-In-The-Middle* (MITM, *Hombre En El Medio*).

Es simplemente imposible evitar que alguien pueda capturar los paquetes de una red, por tanto la solución más aceptada no es eliminar al MITM, sino hacer que sea irrelevante.

En especial queremos que cualquier dato obtenido por el MITM sea completamente inutilizable puesto que no es el MITM quien presenta un riesgo en sí, sino lo que sería capaz de hacer con la información que obtiene. Es por ello que todos los esfuerzos se deben centrar en hacer que la información sea inútil para cualquier MITM presente.

Por lo general, durante el proceso de reforzar la seguridad en Caronte siempre se debe tener en cuenta que existe al menos un Man-In-The-Middle (al que llamamos David).

Técnicas de ofuscación de código y datos suelen ser muy importantes aquí.

Un ejemplo que hace Caronte para evitar que la información robada por un MITM sea de utilidad es ofuscar el identificador del usuario.

Generalmente los usuarios se identifican usando un nombre único, un email, un teléfono, etc. El usuario envía su email para decir quien es y luego hace uso de su contraseña para verificar que es realmente quien dice ser.

En Caronte sin embargo se usa el email de manera indirecta. Dicho de otra forma, se usa un texto derivado del email (calculado mediante una operación matemática irreversible sobre el email). De esta forma, aunque un MITM puede robar el identificador de un usuario, nunca será capaz de averiguar la verdadera identidad del usuario (su email); información que suele ser muy valiosa (robo de identidad, ataque por fuerza bruta, etc).

De la misma forma, aunque el MITM robe el ID del usuario y lo use, no sería capaz de descifrar el TGT para convertirlo en un SGT, puesto que desconoce la contraseña del usuario.

## 7.2. Ataque tipo *Replay*

El ataque más común realizado por un MITM. Los ataques tipo *replay* consisten en la inyección en la red de paquetes que ya han circulado antes con la intención de reusar la información emitida verídicamente por el usuario, haciéndose pasar por el emisor original del mensaje.

Es fácil entender por qué es un ataque tan crucial de solventar. Si un MITM se hace con un paquete conteniendo un *ticket*, puede usar este paquete para emitir el *ticket* una y otra vez enmascarando al emisor original.

Como hemos visto en el apartado sobre Man-In-The-Middle, es prácticamente imposible determinar si alguien está espiando los paquetes de la red, y con mucha seguridad se puede afirmar que siempre habrá un MITM y que siempre robará todos los paquetes de la red.

Por tanto podemos afirmar que cualquier *ticket* emitido por el usuario en un tiempo  $t$  será observado y copiado por un MITM, así como los *tickets* generados anteriormente (desde 0 hasta el presente).

La forma de solucionar el problema radica en invalidar todos los *tickets* que han sido generados antes del *ticket* actual. Suponiendo un *ticket* actual  $t$ , todos los *tickets* anteriores a este ( $t-1$ ,  $t-2$ ,  $t-3$ ,  $t-4$ , ..., 0) son inválidos; sólo el *ticket* actual ( $t$ ) es válido. Cualquier *ticket* sucesor ( $t+1$ ) automáticamente invalida al anterior.

El protocolo Kerberos resuelve este problema mediante *timestamps*: cuando Kerberos envía el TGT al usuario, este contiene un *timestamp* del momento exacto en el que se creó. Cuando el usuario descifra el TGT y lo convierte en un SGT, el *timestamp* ha cambiado, reflejando un instante de tiempo más avanzado ( $t+1$ ) y por tanto invalidando los anteriores ( $t$ ,  $t-1$ ,  $t-2$ , ..., 0).

Sin embargo hemos visto en secciones anteriores que esta solución no es viable si los dos sistemas no tienen sus relojes perfectamente sincronizados, lo cual es el caso que se da en Internet. Simplemente es imposible pedirle a todos los usuarios del mundo que tengan su reloj sincronizado perfectamente con el de un servidor central.

La solución que propone Caronte es mucho más simple: en lugar de un *timestamp* se hace uso de un contador que se encuentra dentro del *ticket*. Cuando Caronte emite el TGT, este contador está inicialmente a 0. Cuando el usuario emite su SGT, el contador se incrementa en 1, por tanto cualquier *ticket* con un contador inferior al que se espera Caronte queda descartado e invalidado. En todo momento el contador de *tickets* de

Caronte debe estar sincronizado con el del usuario, lo cual es bastante más fácil de hacer; el cliente incrementa el contador cada vez que emite un *ticket* y Caronte hace lo mismo cada vez que recibe un *ticket*.

Si en algún momento el MITM obtiene un SGT del usuario, no será capaz de volver a usarlo puesto que el contador de este *ticket* ya no está en sincronía con el de Caronte. Sólo el usuario que posee el TGT descifrado (información para crear el SGT) y el contador actual correspondiente será capaz de crear un SGT válido para Caronte.

Cuando Caronte detecta el uso de un *ticket* que ya no es válido (el contador es inferior al de Caronte) se determina que hay un intento de ataque por *replay* y Caronte procede a invalidar por completo la sesión y a forzar a que el usuario establezca una nueva sesión (con nuevos *tokens* y contador reiniciado).

El único problema con este método es que si se pierde de manera natural un *ticket* en la red de camino a Caronte, los contadores del usuario y de Caronte no estarán sincronizados y se debe restablecer la sesión. También elimina la posibilidad para un usuario de usar el mismo *ticket* para identificarse con varios proveedores a la vez, en este caso el usuario debe identificarse secuencialmente, de uno en uno.

Aunque pueda parecer excesivo tener que iniciar sesión cada vez que se pierde un *ticket*, por lo general no sueles necesitar más de dos *tickets* para establecer una comunicación entre Alice y Bob: el primer SGT que Alice envía a Caronte para confirmar y el segundo SGT que envía Alice a Bob para establecer una sesión. Por lo general el contador de un *ticket* no va a superar el número 2 a la hora de hacer una comunicación entera (incluyendo el establecimiento de la clave secreta entre Alice y Bob).

Todo esto se traduce a que la probabilidad de perder un *ticket* es baja puesto que no hay muchos en circulación.

### 7.3. Ataque *Selfie*

Aunque es un ataque poco conocido ha demostrado ser letal. Descubierta por primera vez en TLS v1.2 (y posteriormente corregido en TLS v1.3), el ataque *Selfie* consiste en usar las propias credenciales de un usuario contra este mismo.

De manera general:

- Alice le pide sus credenciales a Bob (certificado en caso de TLS, *ticket* en caso de Caronte). Bob envía sus credenciales a Alice.
- Bob le pide sus credenciales a Alice, pero Alice responde con las credenciales del propio Bob. Como Bob solo verifica que las credenciales sean correctas, pero no de quién son, se establece una comunicación entre Alice y Bob, sin que Bob sepa que en ningún momento Alice se ha autenticado como tal.

Esta vulnerabilidad es crítica en el proceso de creación del KGT, puesto que Bob necesita el SGT de Alice para construir el KGT, Alice puede simplemente responder enviando el propio SGT de Bob. La solución es hacer que Caronte verifique que el KGT contiene el SGT de dos usuarios diferentes. Es también importante que el proveedor de servicios (Bob) no envíe nunca su SGT al usuario (Alice).

#### 7.4. Ataques por fuerza bruta

Otra técnica a conocer que ha resultado ser efectiva en incontables ocasiones. Los ataques por fuerza bruta consisten en probar todas las posibles combinaciones que pueda tener una clave hasta dar con la correcta.

Cuando hablamos de contraseñas solemos referirnos a un ataque derivado: el ataque por diccionario, que consiste en usar un diccionario de palabras comunes para ir generando posibles contraseñas. Este ataque es cada vez más sofisticado, haciendo uso de técnicas de ingeniería social para agilizar los resultados (palabras más comunes, nombres propios, nombres de películas, libros y demás información extendida en la sociedad).

Un ejemplo real que se conoce fue un ataque por fuerza bruta sobre la pregunta de seguridad para recuperar una cuenta.

Muchos usuarios decidieron escoger como pregunta de seguridad:

¿Cuál es tu libro favorito?

Aunque a priori parece una pregunta que solo se puede averiguar si se conoce en persona al usuario, la realidad dista mucho pues la mayoría de usuarios optaron por tener la misma respuesta: La Biblia. Respuesta fácil de averiguar debido a que es el libro más vendido en la historia de la humanidad y las probabilidades de que una persona en concreto tenga este libro como su favorito son muy altas.

Los ataques por fuerza bruta no se pueden contrarrestar, pero si que podemos hacer que sea extremadamente trabajoso llevarlos a cabo. La idea es simple: cuantas más posibles combinaciones, y cuanto más lento el proceso de verificar una combinación, más inefectivo se vuelve este tipo de ataques.

Un ataque de fuerza bruta se puede contrarrestar si consideramos lo siguiente:

- El tiempo necesario para romper una clave excede el tiempo de vida útil de la información a descifrar.
- El coste computacional necesario para romper una clave excede el coste de la información a descifrar.

En Caronte los TGT, SGT, KGT y prácticamente todos los demás paquetes intercambiados entre las diferentes entidades, están cifrados usando una clave simétrica de 256 bits. Para “romper” el cifrado de un *ticket* hay que averiguar una clave entre  $2^{256}$  (aprox.  $10^{78}$ ) posibles combinaciones. Si pudiéramos probar un millón de combinaciones de claves por segundo, tardaríamos unos  $10^{71}$  segundos ( $10^{63}$  años) en averiguar la clave. En contraste, el universo tiene una vida estimada de  $4.3 \cdot 10^{17}$  años.

### 7.4.1. Ataques por diccionario

Otra historia completamente diferente ocurre con las contraseñas, puesto que no tienen un tamaño exacto conocido. Una contraseña lo suficientemente larga (entre 8 y 16 caracteres) puede llegar a ser (en teoría) igual de segura que una clave criptográfica de 128 bits.

El problema aquí es que a diferencia de las claves criptográficas donde la distribución de ceros y unos no sigue un patrón específico (es homogéneo), en las contraseñas hay un patrón claro: palabras y dígitos que entiende un usuario. Es por ello que se puede limitar la búsqueda solo a aquellas contraseñas que contengan palabras o secuencias de números, reduciendo drásticamente el espacio de búsqueda.

Esto es exactamente lo que hace un ataque por diccionario. Al igual otros ataques por fuerza bruta, este tampoco se puede evitar con total certeza, pero sí que podemos aumentar la complejidad, tanto espacial como temporal, de un posible ataque por fuerza bruta.

La solución a la que se ha llegado parece trivial, pero no lo es. Empezamos por eliminar el mayor obstáculo: el espacio de búsqueda. El atacante ha reducido el espacio de búsqueda deduciendo que solo aquellas claves que forman palabras y/o dígitos legibles por humanos son potenciales contraseñas, descartando el resto. Para contrarrestar este ataque basta con reemplazar el uso de la contraseña original por una derivada en la que la distribución de ceros y unos es más uniforme, por supuesto unidireccional para que el atacante no pueda inferir la contraseña original a partir de la derivada.

Sin embargo, crear una contraseña derivada no es tan trivial como parece en un primer momento. La solución estándar que proporciona Kerberos, así como el login de Linux y otros muchos sistemas, es el uso de una función *hash*. De esta forma lo que se almacena como la contraseña del usuario no es la contraseña en sí, sino un *hash* de la misma. Kerberos permite usar las funciones hash MD4, MD5, SHA1 y SHA2, entre otros, siendo MD5 la más común. Pero esto todavía no resuelve nada.

Uno de los problemas de usar una función *hash* para almacenar contraseñas son las colisiones. Algoritmos como MD4 y MD5 tienen conocidas vulnerabilidades que generan colisiones. En esencia estamos dando al atacante más posibles combinaciones por cada *hash* almacenado, y por ende más probabilidades de averiguar la contraseña original, o al menos una contraseña espejo. Aunque poco probable y se puede resolver usando funciones *hash* más modernas y de mayor tamaño, resistentes a colisiones.

Por otro lado, el mayor problema de las funciones *hash* en cuanto a este tema se refiere lo podemos encontrar en una de sus principales características: su velocidad.

Es crucial poder ralentizar al atacante; cuanto más lento y tedioso son los cálculos, menos margen de tiempo para completar exitosamente el ataque y más costoso llevar a cabo dicho ataque.

Es aquí donde entran en juego lo que se conoce como *Key Derivation Function* (KDF, *Función de Derivación de Claves*). En términos simples, la KDF nos permite transformar una clave (generalmente una contraseña) de tamaño variable en otra clave criptográfica (también de tamaño variable), cumpliendo la unidireccionalidad, solventando el problema de las colisiones (siempre que se cumpla que el tamaño de la clave derivada sea mayor que la original).

Mediante el uso de iteraciones (también variable), se puede ralentizar el costo operacional de derivar la contraseña, y por tanto ralentizando cualquier ataque por fuerza bruta. Las iteraciones se hacen de tal manera que cada iteración  $i$  depende del cálculo de la  $i-1$ , eliminando la posibilidad de cálculo en paralelo.

El siguiente pseudocódigo es un ejemplo simplificado de una KDF.

```
unsigned char[] KeyDerivationFunction(unsigned char[] input, int iterations){  
    unsigned char[] temp = input;  
    for (int i=0; i<iterations; i++){  
        temp = HashFunction(input+temp)  
    }  
    return temp;  
}
```

La función estándar usada y estudiada más ampliamente en la actualidad es *Password Based Key Derivation Function 2 (PBKDF2)*. El código anterior está basado en la primera versión, quitando el cómputo necesario para producir un resultado de tamaño variable. Otro algoritmo comúnmente usado es *bcrypt*.



#### 7.4.2. Ataques por fuerza bruta inversa

Otro tipo de ataque, aunque poco común, ha demostrado ser bastante eficaz.

El ataque por fuerza bruta se basa en la premisa de que, debido a fenómenos socioculturales, existen una serie de contraseñas que tienen una alta probabilidad de existir, dicho de otra forma, muchos usuarios comparten contraseñas iguales o similares.

En este tipo de ataque se hace uso de dos diccionarios, uno para generar contraseñas con alta probabilidad de existir, y otro para generar emails (o nombres de usuarios).

Estamos ante un caso en el que el atacante intenta averiguar un email vulnerable a partir de contraseñas conocidas.

El problema con este ataque lo encontramos cuando el servidor de autenticación muestra un mensaje de error al rechazar un email que no encuentra en la base de datos, lo cual facilita al atacante identificar emails válidos de aquellos inválidos.

La solución de Caronte es la introducción del *ticket falso*. Cada vez que el servidor de autenticación recibe un identificador de usuario (ya sea email o derivado) que no se corresponde con ningún usuario conocido, el servidor responde creando un *ticket* que es virtualmente idéntico a cualquier TGT de un usuario real, pero la información contenida es inválida y la clave de cifrado es conocida sólo por Caronte. El atacante es incapaz de saber si el TGT recibido por Caronte es verdadero (correspondiente a un email que existe) o, de lo contrario, es un *ticket* indescifrable correspondiente a una cuenta inexistente.

## 7.5. Ataques específicos de Kerberos

No todos los ataques a los que nos enfrentaremos son ataques genéricos derivados de diversas ramas de la ciberseguridad. De hecho, una serie de ataques han sido diseñados exclusivamente para atacar el protocolo Kerberos, los cuales tenemos que estudiar y evitar su propagación a Caronte.

Teniendo en cuenta que Caronte es una evolución del protocolo Kerberos, podemos afirmar que estos ataques serán también enfocados contra Caronte.

Los principales ataques a los que se enfrenta Kerberos son *Pass The Ticket* y *Golden Ticket*.

Primero veamos el menos complejo: *Golden Ticket* (existen derivados conocidos como *Silver Ticket*).

Un *Golden Ticket* es un TGT especial que otorga al usuario portador de acceso privilegiado al sistema. En Kerberos, el TGT contiene información sobre los permisos de acceso que tiene el usuario que se identifica sobre el servicio que requiere.

Esta información de acceso cifrada usando una clave maestra de Kerberos. Si esta clave maestra es robada, un atacante puede generar TGTs válidos con la información de acceso que deseada. Toda la seguridad de los TGT radica en el secreto de la clave maestra.

Resolver este tipo de ataques no es trivial. El protocolo Caronte simplemente omite la información sobre accesos y privilegios, lo cual es delegado principalmente al proveedor de servicios y la información que este posee sobre el usuario que se identifica. Aparte, un TGT en Caronte es cifrado usando la propia contraseña del usuario, en lugar de una contraseña maestra, y de por sí el TGT no tiene funcionalidad propia más allá de contener la información que el usuario necesita para crear los SGT. Los tickets en Caronte simplemente no sirven para otorgar privilegios al usuario, solo para identificarlos. Es tarea del proveedor de servicios contener y procesar la información sobre los privilegios de un usuario una vez se establece una sesión.

El otro tipo de ataque es mucho más crucial: *Pass The Ticket*.

Este tipo de ataque es un derivado de Man-In-The-Middle en el que un usuario malicioso roba el SGT al usuario real y lo presenta como suyo. Ya que hablamos de una conexión con tráfico abierto, es imposible saber si el SGT que nos ha llegado procede realmente del usuario original. Podemos saber con certeza que el usuario ha creado y emitido el *ticket*, pero no podemos saber si existe un MITM the consumirá los recursos del usuario.

Hay varias medidas que Caronte toma en cuenta para prevenir este tipo de ataques:

- El SGT enviado por el usuario al proveedor de servicios debe ser verificado por Caronte. En el protocolo Kerberos el proveedor nunca verifica el SGT del usuario.
- No se proporciona una clave de sesión hasta que Caronte no verifique el SGT del usuario por parte del proveedor (se verifica el KGT).
- Caronte no proporciona métodos para controlar la autorización, sólo la autenticación. Es el proveedor quien debe proporcionar medidas para garantizar la autorización del usuario, a ser posible mediante una tecnología estándar tal como lo es OAuth2.

## 7.6. Ataques de inyección de código

Hasta ahora hemos visto los principales tipos de ataque a los que se enfrenta Caronte en el proceso de autenticación y establecimiento de comunicación.

No obstante, no todos los ataques están relacionados con el robo de credenciales.

Existen una serie de ataques cuyo objetivo es el de ejecutar código malicioso en el servidor, haciendo uso de vulnerabilidades en las librerías para interpretar archivos de texto o en el procesamiento de *input* del usuario.

Algunos ejemplos según su área de ataque son:

- Inyección de código en las URLs: Path Traversal Attack, XPATH Injection
- Inyección de código en la máquina del cliente: Cross-Site Scripting, DOM Injection, Cross-Site Request Forgery (CSRF).
- Inyección de código en los formularios y demás *inputs* del usuario: SQL Injection, Eval Injection.
- Inyección de código en archivos malformados: CSV Injection, Unicode Encoding, XML o JSON malformados.

De por sí, estos ataques se escapan del alcance del proyecto Caronte, es por ello que no se definen técnicas ni medidas en el protocolo contra este tipo de ataques, pues se consideran ataques generales que pueden afectar a prácticamente cualquier servidor o aplicación web, no sólo al proceso de autenticación. En muchos casos estos ataques se llevan a cabo por usuarios que ya están verificados en el sistema de manera “legal”.

Evitar estos ataques requiere de un buen diseño en la plataforma web que haga uso de tecnologías para contrarrestar estas vulnerabilidades. En el proyecto de ejemplo de Caronte que se propone hace uso de Django (y Django REST) Framework, el cual ofrece una gran defensa contra muchos de estos ataques (especial mención a las librerías DAO de Django que protegen contra ataques de SQL Injection). Otras herramientas existen tales como *Hibernate* para Java. La aplicación por parte del cliente debe también estar ejecutando en un entorno seguro. Recordemos que Caronte no se presenta como una alternativa a las medidas ya existentes, sino como un añadido. Cuanta más protección mejor. Se recomienda el uso de navegadores modernos y actualizados que ofrezcan una defensa contra *malware* que ataquen el computador del usuario.

## **8. Proyecto Caronte**

A modo práctico se presenta un proyecto software que implementa el protocolo Caronte como servidor de autenticación mediante API REST.

El servidor en sí está compuesto por tres componentes principales:

- El servidor REST que proporciona las primitivas de autenticación. Contiene la lógica del servidor Caronte en sí.
- Las librerías del cliente proporcionan la lógica que necesita el usuario para comunicarse con Caronte y hace efectiva la autenticación. Esta librería será usada por clientes de Caronte, es decir, tanto por los usuarios como por los proveedores.
- La interfaz de seguridad y criptografía. Una pequeña librería que enmascara las operaciones referentes a primitivas de seguridad (hashing, PRNG, KDF, encrypt/decrypt, etc.), bajo una interfaz común tanto para Caronte como para los clientes. Esta interfaz es responsable de mantener las medidas de seguridad actualizadas sin afectar a la lógica interna del protocolo. Si en un futuro es necesario cambiar un algoritmo criptográfico específico, este cambio será transparente para Caronte y no afectará el funcionamiento del protocolo en sí.

Veremos en más detalle cómo se han implementado estos tres componentes, pero antes veamos como configurar y ejecutar el servidor y el cliente de prueba.

## 8.1. Instalar y ejecutar el proyecto

El servidor de autenticación Caronte está escrito usando Django, un framework web ampliamente conocido que proporciona una elegante solución para servidores web.

Más concretamente nos interesa usar los componentes REST y la potente herramienta para crear DAOs y acceder a bases de datos.

Los pasos a seguir son triviales: instalar Python 3 y Python-PIP (gestor de dependencias de Python), Django REST Framework y las librerías criptográficas para luego desplegar el servidor como cualquier proyecto Django estándar.

A continuación un ejemplo para el Sistema Operativo Ubuntu en una instalación limpia, pero los pasos a seguir son virtualmente iguales en otros sistemas, sustituyendo el método de instalar Python 3 y Python-PIP.

Primero debemos asegurarnos que tenemos la última versión de Python 3 y Python-PIP:

```
~$ sudo apt-get install python3 python3-pip
```

La herramienta Python-PIP es un gestor de dependencias de Python que nos permite localizar, descargar e instalar paquetes Python para usar directamente en nuestro proyecto.

El proyecto se ha escrito con el mínimo posible de dependencias para facilitar su instalación y uso. Esto significa que algunos aspectos como la base de datos usan las librerías estándar. En el caso de la BBDD, se usa el motor de SQLite3. Cambiar a otro motor (PostgreSQL, MySQL, MongoDB, etc) es trivial y está muy documentado, pero añade dependencias que no nos interesa abordar en este proyecto. Es por eso que sólo hay dos dependencias principales: Django y PyCrypto.

Para Django necesitaremos tanto el framework base como el REST framework y los *CORS Headers* (permitir que apps de otro origen se comuniquen con Caronte).

Lo podemos instalar todo con el siguiente comando:

```
$ sudo pip3 install django.djangorestframework djang-cors-headers pycrypto
```

Una vez tenemos todas las dependencias instaladas, procedemos a configurar y desplegar el servidor. En la raíz del proyecto encontraremos el archivo `manage.py` que nos permite realizar operaciones de mantenimiento y despliegue.

Debemos asegurarnos que la base de datos está construida y sincronizada con los DAOs de Django. Ejecutaremos las siguientes instrucciones en la raíz del proyecto:

```
$ python3 manage.py makemigrations caronte_server
```

```
$ python3 manage.py migrate
```

Sin entrar mucho en detalles, la primera sentencia genera *scripts* de Python con las sentencias SQL para crear la base de datos y la segunda sentencia aplica dichos *scripts*.

Una vez tenemos la base de datos sincronizada ya podemos ejecutar el proyecto usando la siguiente sentencia:

```
$ python3 manage.py runserver
```

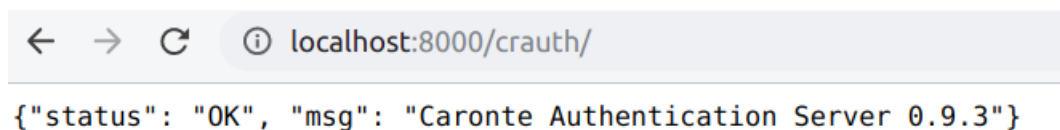
Si todo está correcto deberíamos ver la siguiente pantalla:

```
Performing system checks...

System check identified no issues (0 silenced).
August 31, 2019 - 12:34:21
Django version 2.2.4, using settings 'caronte.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Para comprobar que el servidor está ejecutando correctamente vamos a hacer un test sobre la API REST.

Si realizamos una petición HTTP GET sobre la URL `/crauth/` obtendremos una respuesta en formato JSON con el nombre y la versión de Caronte. Podemos comprobar esto en el navegador:<sup>3</sup>



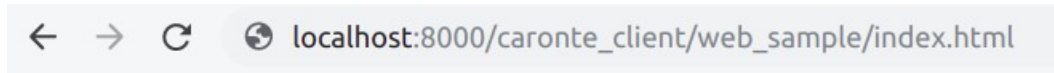
The screenshot shows a web browser address bar with the URL `localhost:8000/crauth/`. Below the address bar, the JSON response is displayed: `{"status": "OK", "msg": "Caronte Authentication Server 0.9.3"}`.

3 En la versión de prueba (modo *DEBUG*) se crearán los usuarios Test y Provider que nos servirán para depurar el servidor Caronte.

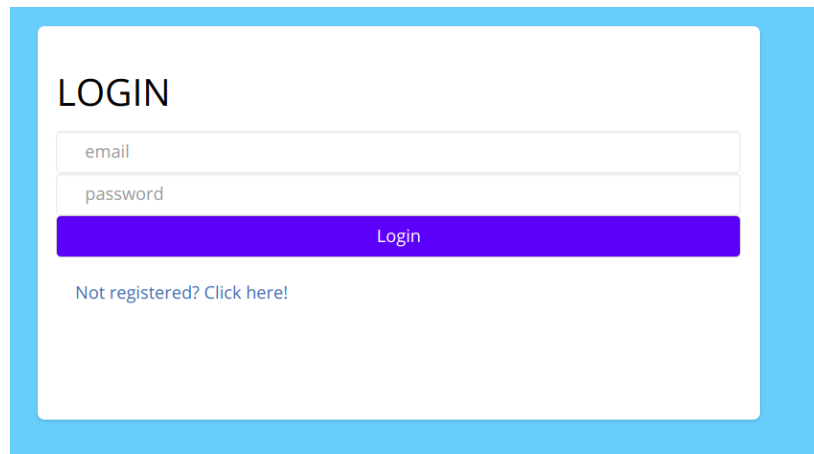
## 8.2. Ejecutar la aplicación de prueba

El proyecto Caronte dispone de una página web de prueba y ejemplo.

Para iniciar la plataforma acudiremos a la siguiente dirección:



En la cual nos encontraremos la pantalla de login que vemos en la figura 8.1:

A screenshot of a login form. The form is titled 'LOGIN' in bold. It contains two input fields: 'email' and 'password'. Below these fields is a blue button labeled 'Login'. At the bottom of the form, there is a link that says 'Not registered? Click here!'. The entire form is set against a light blue background.

*Figura 8.1*

Introducimos las credenciales para el usuario test:

- email: [test@caronte.com](mailto:test@caronte.com)
- contraseña: Caront3Te\$t

Y se nos mostrará la pantalla principal (figura 8.2):



| User Name      | e-mail           | Join Date                        |
|----------------|------------------|----------------------------------|
| Caronte Tester | test@caronte.com | 2019-08-31 12:56:57.602286+00:00 |

Connected to: Caronte Authentication Server 0.9.3  
Ticket Validates: Success!  
Service Provider Data:

Validate Ticket

Invalidate Ticket

Revalidate Ticket

Connect to Service Provider

Logout

Update User Info

Caronte Tester

old password

new password

Update

Figura 8.2

Veamos el significado de cada apartado.

En primer lugar nos encontramos con una serie de datos sobre el usuario que ha hecho login.

| User Name      | e-mail           | Join Date                        |
|----------------|------------------|----------------------------------|
| Caronte Tester | test@caronte.com | 2019-08-31 12:56:57.602286+00:00 |

Esta información es todo lo que conoce Caronte sobre el usuario (quitando contraseñas y demás datos sensibles). El protocolo Caronte no define realmente esta funcionalidad puesto que es responsabilidad del proveedor de servicios dotar al usuario de información sobre su cuenta una vez establecida la clave de sesión. De hecho no se define en sí un identificador de usuario (puede ser un email, NIF, teléfono, etc). El identificador ni siquiera se usa directamente en la comunicación, sino un texto derivado de este.

Pero a efectos prácticos, y porque no supone ninguna comunicación extra, Caronte proporciona al usuario estos detalles como respuesta a recibir el primer SGT válido (recordemos que antes de que el usuario pueda hacer uso de su SGT debe validarlo una vez con Caronte).

En un entorno de producción real sin embargo debe ser el proveedor de servicios quien provea de los datos al usuario.

La siguiente fila de texto se nos muestra el estado actual de la sesión con Caronte y con el proveedor de servicios.

```
Connected to: Caronte Authentication Server 0.9.3
Ticket Validates: Success!
Service Provider Data:
```

Seguido de los respectivos botones para interactuar con la sesión (figura 8.3).

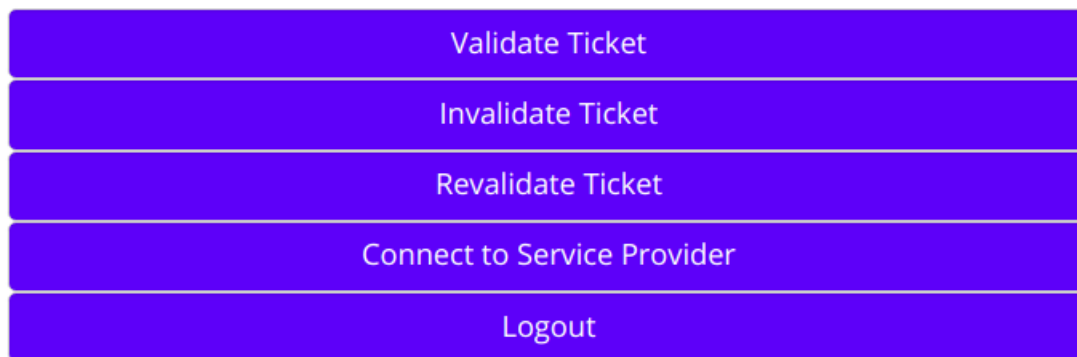


Figura 8.3

En orden de apariencia los botones tienen la siguiente funcionalidad:

- Validate Ticket: envía el SGT a Caronte para ser validado. Caronte devuelve un mensaje de OK o ERROR respectivamente. Cada vez que se crea un nuevo *ticket*, el usuario incrementa el contador para ese *ticket*, el cual es verificado por Caronte y actualizado para evitar ataques *replay*.

- Invalidate Ticket: esta función envía a Caronte un *ticket* intencionadamente incorrecto para invalidar por completo cualquier *ticket* creado usando la sesión actual.

```
Connected to: Caronte Authentication Server 0.9.3
Ticket Validates: Error
Service Provider Data:
```

Para hacer esto el *ticket* que se envía tiene su contador desactualizado (básicamente el propio usuario hace un ataque *replay* enviando un *ticket* que ya se ha enviado, efectivamente invalidando todos los *tickets* de esta sesión):

```
[31/Aug/2019 13:44:06] "POST /validate/ HTTP/1.1" 200 42
ERROR: pausable replay attack on user <test@caronte.com>, ticket count does not match, expected 3, got 0
ERROR: user <test@caronte.com> verifies with wrong ticket
```

- Revalidate Ticket: con esta función podemos revalidar la sesión en caso de que nuestros *tickets* ya no validen con Caronte. La función realmente establece una nueva sesión con Caronte de la misma forma que lo haría el *login*, pero debido a que ya teníamos una sesión anterior hay ciertos datos que ya conocemos (email, password y algunas claves derivadas).

- Connect to Service Provider: con esta función podemos simular el proceso de *login* sobre una aplicación web. En este proceso el usuario envía su SGT que ha generado con Caronte para identificarse (en este caso el *ticket* reemplaza al email y contraseña). El proveedor de servicios transforma el SGT en un KGT y lo envía a Caronte para validar.

Estando todo correcto, Caronte responde proporcionando las claves de sesión cifradas que necesitan el usuario y el proveedor de servicios para comunicarse. Una vez el usuario recibe confirmación de que el *login* ha sido exitoso podrá usar la clave de sesión para cifrar y descifrar la comunicación con el proveedor de servicios, que también dispondrá de esta clave secreta.

```
Connected to: Caronte Authentication Server 0.9.3
Ticket Validates: Success!
Service Provider Data: super secret information not accessible without a login
```

Por último podemos encontrar un formulario para cambiar las credenciales del usuario. Caronte permite que el usuario pueda incrustar datos dentro dentro del SGT que serán cifrados junto al resto de información. Es aquí donde el usuario puede seguramente transmitir a Caronte que desea cambiar de contraseña de ahora en adelante (figura 8.4).

Update User Info

|                |
|----------------|
| Caronte Tester |
| old password   |
| new password   |
| Update         |

Figura 8.4

Debido a la naturaleza de los *tickets*, éste cambio de contraseña es instantáneo y no requiere de un *re-login* para hacer efectivo. Una vez Caronte verifica el *ticket* y procede a cambiar la contraseña del usuario, en su respuesta ofrecerá los datos que el cliente necesita para calcular la nueva contraseña derivada. Hablaremos sobre el funcionamiento interno de los *tickets* y las contraseñas más adelante.

Otro aspecto que hay que hablar sobre Caronte es el registro de usuarios, o más bien, la falta de soporte para hacer esto. El protocolo funciona bajo la base de que Caronte ya conoce las credenciales de los usuarios involucrados, pero no se responsabiliza de cómo se han establecido estas credenciales. Esto es principalmente debido a que el establecimiento de claves es otro tema aparte y hay muchas soluciones muy variadas e igual de válidas; podemos hacer que Caronte genere una contraseña aleatoria y la envíe mediante un servidor de correos seguro, podemos hacer que el proceso de registro funcione bajo HTTPS, podemos tener a una persona física encargada de administrar la BBDD y proporcionar contraseñas, ... en fin todo va a depender realmente del escenario en el que se está usando Caronte. Una opción (de las muchas que hay) que se muestra en el ejemplo web (figura 8.5) es el uso de una clave maestra para cifrar el proceso de registro, clave que solo conocerán los administradores del sistema, quienes se encargarán de registrar a los usuarios.

|                 |
|-----------------|
| retype password |
| server secret   |
| Register        |

Figura 8.5

### 8.3. Análisis de la prueba

Para terminar con la interfaz gráfica vamos a hacer una observación sobre el proceso de *login* con Caronte y con el proveedor de servicios analizando el tráfico de paquetes usando el inspeccionador de red de Chrome.

La figura 8.6 corresponde a todas las llamadas que el cliente hace al servidor desde el momento en el que el usuario escribe su email y contraseña hasta que se muestra la pantalla principal.

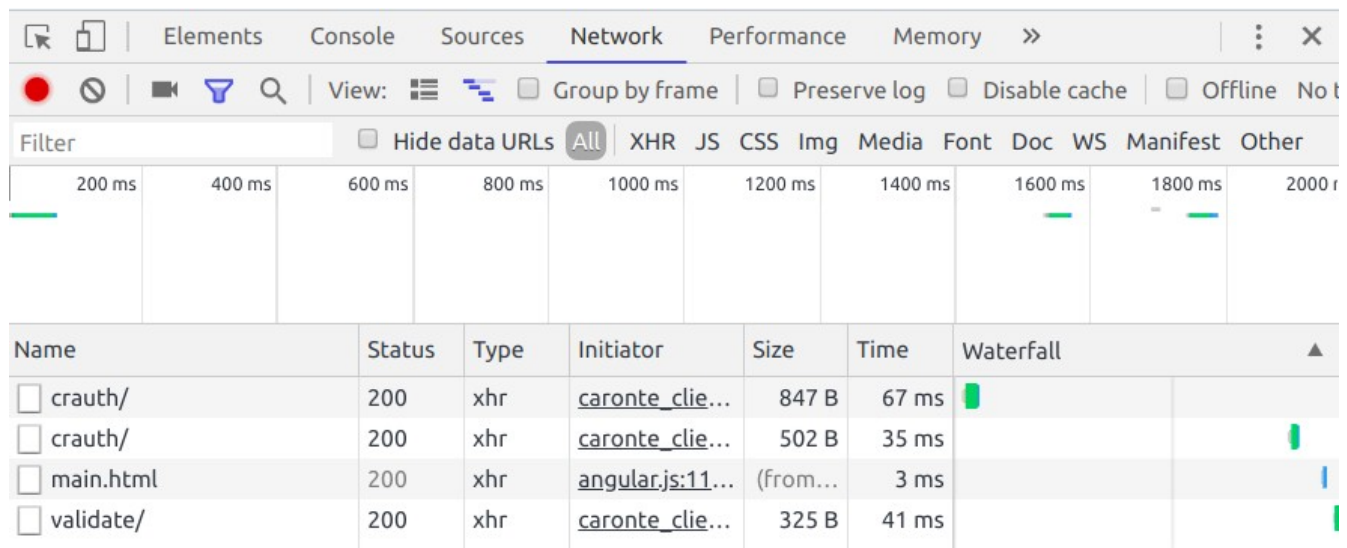


Figura 8.6

De especial interés son las dos primeras llamadas a la URL *crauth* de la API REST, que son realizadas previo a cargar la página principal (*main.html*), lo cual nos da una pista de que las llamadas son el proceso de *login*.

Entramos sobre la primera llamada y podemos encontrar el siguiente mensaje destinado al método POST de la URL *crauth*:

▼ Request Payload    view source

```
{ID: "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnI1ie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A=="}
ID: "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnI1ie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A=="
```

Lo primero que llama la atención es que la llamada a esta API por parte del cliente sólo tiene un parámetro: el ID del usuario (o mejor dicho, un texto derivado del email). El usuario no envía en ningún momento su contraseña pues no lo necesita.

Para probar su identidad solo debe ser capaz de descifrar el TGT y transformarlo en un SGT. Se hace uso de un texto derivado del email para enmascarar la verdadera identidad del usuario, pero también para ralentizar ataques por fuerza bruta inversa. La respuesta a esta llamada es mucho más peculiar y contiene información crucial en el proceso de autenticación.

```
▼ {status: "OK", IV: "kkruI4F1Vt++hwtqY+/UCBK7m7SukM1PjCarpI7M=", pw_iters: 1000,...}
  IV: "kkruI4F1Vt++hwtqY+/UCBK7m7SukM1PjCarpI7M="
  TGT: "6XRaulqG6DIwI3K6U2kLF2TK+c+P5ftHcaFbbAlPiaBpFEI4JZKj+ubutqYTWtmDVouCxn/IZyQYa6l
  pw_iters: 1000
  status: "OK"
  tgt_iv: "4HHc4ipPZVNB7/0F0gAStA=="
```

Los datos que recibimos del servidor tienen la siguiente información:

- status: siempre será “OK” en el caso del *login*, incluso si el email derivado es incorrecto (ver apartado de ataques por fuerza bruta inversa).
- IV: un vector de inicialización que se ha usado para derivar la contraseña del usuario. La contraseña del usuario no se deriva directamente en una clave criptográfica, sino que se transforma usando un proceso idéntico al email derivado, en el que el texto resultante es de un tamaño mayor que el original, eliminando colisiones presentes en funciones *hash*, pero conservando la unidireccionalidad. El IV es necesario para derivar correctamente la contraseña usando una KDF para ser convertida en una clave criptográfica. Este IV está cifrado usando otra contraseña derivada de manera estática.
- pw\_iters: número entero positivo que determina el número de iteraciones a realizar para derivar la contraseña y la clave criptográfica.
- tgt\_iv: el vector de inicialización usado para cifrar el TGT.
- TGT: el *Ticket Granting Ticket* en sí. Está cifrado usando una clave criptográfica derivada de la contraseña derivada.

El usuario realiza los siguientes pasos:

- Genera una contraseña derivada estática (*p1*) desde la contraseña original.
- Descifra el IV usando *p1*.
- Usa el IV y la contraseña original para generar la contraseña derivada dinámica (*p2*).
- Usa *p2* para descifrar el TGT y obtener una clave temporal.
- Usa la clave temporal para cifrar el SGT.

Una vez generado el SGT, el usuario debe validarlo por primera vez con Caronte antes de poder usarlo para autenticarse con el proveedor de servicios. Es por eso que hay una segunda llamada a la URL *crauth*. Esta segunda llamada a *crauth* se hace sobre el método PUT, en el cual el usuario envía su primer SGT a Caronte para que este confirme y dé por válida la sesión.

▼ Request Payload    view source

```

{
  "ticket": {
    "ID": "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnIlie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A==",
    "ID": "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnIlie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A==",
    "IV": "xCgWt1RRJx+r1xFEq9zCmg==",
    "SGT": "XXzXeRtXUioQjJGk0h84uR3nLL+hac37jY7f5mrVnGdqjYEttkZPI0117ZouDS/gIwTR7Bh/48blJyum+NisBWQRNKhgwd8KkH"
  }
}

```

Los campos que componen el *ticket* son:

- ID: el identificador de usuario, que se corresponde con el vector de inicialización usado para derivar la contraseña.
- IV: el vector de inicialización usado para cifrar el SGT.
- SGT: contiene la información crucial sobre el *ticket* (*token*, contador, etc), cifrado usando una clave temporal encontrada dentro el TGT.

Una vez recibido, Caronte procede a identificar al usuario, descifrar el SGT con la clave temporal y comprobar que los valores son correctos. En este punto Caronte da por válido el ticket y el usuario puede proceder a enviarlo al proveedor de servicios. Es aquí donde nos encontramos con la pantalla principal y nuestro *ticket* perfectamente verificado y listo para su uso. Procedemos a comunicarnos con el proveedor de servicios y observar los componentes de la llamada (figura 8.7).

| Name                               | Status | Type | Initiator                       | Size     | Time   | Waterfall |
|------------------------------------|--------|------|---------------------------------|----------|--------|-----------|
| <input type="checkbox"/> crauth/   | 200    | xhr  | <a href="#">caronte_cli...</a>  | 847 B    | 67 ms  |           |
| <input type="checkbox"/> crauth/   | 200    | xhr  | <a href="#">caronte_cli...</a>  | 502 B    | 35 ms  |           |
| <input type="checkbox"/> main.html | 200    | xhr  | <a href="#">angular.js:1...</a> | (from... | 3 ms   |           |
| <input type="checkbox"/> validate/ | 200    | xhr  | <a href="#">caronte_cli...</a>  | 325 B    | 41 ms  |           |
| <input type="checkbox"/> provider/ | 200    | xhr  | <a href="#">script.js:178</a>   | 943 B    | 549 ms |           |
| <input type="checkbox"/> provider/ | 200    | xhr  | <a href="#">script.js:173</a>   | 443 B    | 48 ms  |           |

Figura 8.7

Para obtener los datos que el usuario requiere del proveedor de servicios, dos nuevas llamadas son realizadas.



Como en cualquier otra aplicación web, antes de que el usuario pueda disponer del servicio, debe primero proporcionar sus credenciales para autenticarse. Es por eso que las dos llamadas corresponden a (1) el proceso de *login* y (2) la consumición del servicio. Los datos que el usuario envía al proveedor son los siguientes:

▼ Request Payload [view source](#)

```
▼ {, ...}
  ▼ ticket: {ID: "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnIlie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A==" , ...}
    ID: "mo4fgQAhex+Nzj8HZI8Gnho6Qi79qJnIlie3jcS570WzW0np1ht7Siq50rzB40YqpNTXn2WU0S9saxhgZJzh0A=="
    IV: "ZbIJisBvIoc9lLaPYCGqmg=="
    SGT: "9HQxCnFlgJV+uEQVGL0zMUzLW5bTY0EC6AH907NyycqX2+30afbRSKlI13wNDQ0afXTvS4wtPhzFl42JdZ0SWhBarfxHbknwnCvd
```

En efecto, el usuario usa el *ticket* de Caronte para identificarse en lugar de usar su email y contraseña. El proveedor de servicios hace uso de las librerías de Caronte para transformar este ticket en un KGT y validarlo (veremos el proceso en detalle más adelante). Una vez hecho, Caronte entrega las claves de sesión al proveedor de servicios, el cual descifra su copia y devuelve al cliente la suya.

La respuesta del proveedor de servicios a un login correcto es la clave de sesión cifrada para el usuario, como podemos observar en la siguiente imagen:

```
▼ {status: "OK", msg: "Tickets verified", ...}
  key: "eyJrZXkiOiAiMWQwdXFSLlhK0XEweDJJcDR0WXh1VFlwWX"
  msg: "Tickets verified"
  status: "OK"
```

La clave está cifrada usando la misma clave temporal que el destinatario usa para cifrar los SGT.

Es entonces que el usuario puede hacer uso de los servicios. En nuestro ejemplo el usuario simplemente hace una petición HTTP GET al proveedor para recibir un mensaje de texto. El proveedor de servicios sólo acepta aquellas peticiones de usuarios *loggeados*, y para los que lo están enviará la información cifrada usando la clave de sesión.

```
▼ {, ...}
  msg: "eyJpdii6ICJqbFV5cGprT3g0OGZvL1lKQVJuaEVBPT0iLCAiZGF0YSI6ICJXd25XTVZ5T2ZzbTdkZ
```

Esta respuesta contiene el mensaje y el IV necesario para descifrarlo.



#### **8.4. Principales componentes software**

Todo proyecto software tiene su inherente proceso de diseño e implementación de la funcionalidad subyacente.

En el proyecto Caronte la funcionalidad está dividida en tres principales módulos: la interfaz criptográfica, el código del servidor Caronte (a su vez compuesto de API REST y DAO) y el código del cliente.

La interfaz criptográfica permite crear una abstracción genérica sobre los procesos criptográficos internos. Nos sirve para enmascarar las librerías criptográficas (que dependen del lenguaje, versión y entorno), pero también para reforzar el uso de librerías seguras y estándar (CryptoJS, PyCrypto, javax.crypto, etc.).

El código del cliente nos permite realizar las operaciones de comunicación con el servidor Caronte para poder generar *tickets*, conectar con proveedores y demás funcionalidad del protocolo.

A su vez, el servidor provee de una API REST estándar para facilitar la conexión con cualquier cliente independiente de su origen y sus componentes software.

La comunicación entre un cliente y Caronte se realiza principalmente mediante *tickets*, los cuales son paquetes de información cifrados con claves criptográficas conocidas por el servidor y el cliente. A continuación veremos su estructura interna.

#### **8.4.1. Estructura interna de los Tickets**

Hemos hablado mucho sobre los *tickets* y su importancia en el proceso de autenticación.

Conocemos también algunos datos como el contador interno, pero hasta ahora no hemos estudiado los componentes de los diversos *tickets* que existen en Caronte.

Recapitulando, sabemos que en Caronte existen cuatro tipos diferentes de *tickets*:

- Ticket Granting Ticket
- Service Granting Ticket
- Key Granting Ticket
- Fake Ticket

Cada uno de estos *tickets* se procesa para ser transformado en el siguiente.

- El usuario recibe un TGT de Caronte y lo transforma en un SGT.
- El proveedor de servicios recibe un SGT del usuario y lo transforma en un KGT.
- El *ticket* especial conocido como *Fake Ticket* no tiene validez ni usabilidad.

Para entender el proceso de transformar un tipo de *ticket* en otro, hay que primero entender qué información contiene cada *ticket*.

#### 8.4.1.1. Ticket Granting Ticket

El primer tipo de *ticket* que nos encontramos, generado como respuesta a una petición de autenticación del usuario.

El TGT tiene dos funcionalidades principales:

1 - Autenticar a Caronte frente al usuario, puesto que el TGT contiene el identificador y número de versión de Caronte cifrado usando la contraseña del usuario. Esto permite al usuario verificar que en efecto se ha comunicado con un servidor Caronte de confianza.

2 – Proporcionar al usuario información para construir el SGT que le identificará.

Toda la información que contiene el TGT viene codificada en un JSON, cifrado usando la contraseña derivada del usuario. Los campos que encontramos en el JSON del TGT se observan en la tabla 8.1.

| Campo              | Significado   |
|--------------------|---|
| Name               | Contiene el nombre del servidor de autenticación (Caronte Authentication Server).   |
| Version            | Contiene la versión del servidor de autenticación Caronte.  |
| Token              | Número especial que identifica el <i>ticket</i> para este usuario. El procedimiento para generar el <i>token</i> no está definido. Este número no debe cambiar nunca para todos los SGT y KGT generados a partir de este TGT. |
| Temp Key (tmp_key) | Clave criptográfica temporal de 256 bits que se usará para cifrar los SGT generados a partir de este <i>ticket</i> .  |

Tabla 8.1

Los campos “name” y “version” servirán al usuario para identificar a Caronte. Gracias a este dato y al hecho de que el TGT está cifrado usando la contraseña derivada del usuario, podemos afirmar que el servidor Caronte que nos ha entregado el TGT efectivamente conoce nuestro usuario y contraseña.

Los campos “token” y “tmp\_key” permitirán al usuario generar un SGT, demostrando a Caronte que el usuario ha descifrado correctamente el TGT y por tanto haciendo efectivo el proceso de autenticación.

#### 8.4.1.2. Service Granting Ticket

Una vez el usuario recibe y descifra el TGT, podrá proceder a construir el SGT que servirá para autenticarse.

Al igual que el TGT, los SGTs contienen su información codificada en JSON y posteriormente cifrada.

Los campos que componen el SGT y su significado son los encontrados en la tabla 8.2.

| Campo   | Significado  |
|---|--|
| Token (t)   | El <i>token</i> procedente del TGT que ha descifrado el usuario. Debe ser copiado del TGT al SGT tal cual.   |
| Counter (c)                                       | Contador de <i>tickets</i> para evitar ataques tipo <i>replay</i> . El primer SGT creado tendrá su contador a 1. Cada SGT sucesor incrementará el contador en 1.   |
| Email, user_iv y otros identificadores de usuario | Información redundante para identificar al usuario.<br>El user_iv ya se envía junto al SGT, pero hay otra copia dentro del SGT así como el email del usuario. Esta información puede ser opcional pero es recomendable.  |
| Datos extra (extra_data)                          | Datos en formato JSON que contienen información adicional y opcional para comunicar al servidor Caronte, cifrados dentro del SGT. No hay un estándar sobre el uso de este campo ni los subcampos que contiene; depende de la implementación de Caronte y las características que este ofrece al usuario. En el proyecto de ejemplo se usa este campo para cambiar la contraseña del usuario (enviando la contraseña antigua y la nueva). |

Tabla 8.2

Observando los campos y su significado, podemos entender mejor el proceso de transformación de un TGT a un SGT.

1 – El usuario descifra el TGT usando su contraseña (derivada) y obtiene el *token* y el *temp\_key*.

2 – Se crea un JSON que contiene el *token* del TGT, un contador inicialmente a 1, los identificadores del usuario (email e IV) y, opcionalmente, los datos extra que sean necesarios.

3 – El usuario cifra el JSON usando la *temp\_key* encontrada dentro del TGT; el SGT ya está listo para ser enviado a Caronte.

### 8.4.1.3. Key Granting Ticket

Finalmente, el usuario envía su SGT al proveedor de servicios, el cual debe realizar otro proceso para transformarlo en un KGT.

Los KGT son más simples que los TGT, conteniendo solamente un campo de información: *Other Ticket*.

Cuando un usuario tiene su SGT cifrado, lo envía junto a su identificador al servidor Caronte para autenticar. Esta llamada es igual para un KGT, pero con una diferencia substancial: el SGT que se envía a Caronte no es el propio, sino el de otro usuario que ha pedido establecer una sesión.

En resumen: Caronte detecta que un usuario ha pedido verificar el *ticket* de otro, y por ende se considera que se establece una comunicación entre ambas entidades (el dueño del *ticket*, y el que ha enviado el *ticket* para verificación). Si el *ticket* valida correctamente, se generan las claves de sesión para ambos usuarios. Es por eso que el proceso de verificar el SGT de otro usuario se conoce como *Key Granting Ticket*.

El proceso de transformar un SGT en un KGT es el siguiente:

- 1 – El usuario Alice envía su SGT a Bob para establecer una conexión.
- 2 – Bob recibe el SGT de Alice y lo vuelve a cifrar usando la clave temporal (campo *temp\_key*) de su propio SGT. A estas alturas el SGT de Alice está cifrado dos veces, una por Alice y otra por Bob. Este nuevo ticket se llamará “KGT”.
- 3 – Bob añade su identificador a este nuevo *ticket* y lo envía a Caronte.
- 4 – Caronte detecta que el *ticket* contiene un “*other ticket*” y lo verifica para generar las claves de sesión.

Es importante saber que un KGT no contiene el SGT del proveedor Bob. No es necesario que Bob añada su SGT puesto que al transformar el SGT de Alice en un KGT ya está añadiendo su identificador y credenciales (el SGT de Alice es cifrado otra vez por la clave de Bob). El SGT de Alice ya contiene la información de seguridad que necesitamos para evitar ataques de tipo *replay*. Añadir el SGT de Bob tendrá efectos negativos, pues en todo momento el contador del SGT de Bob tiene que estar sincronizado con Caronte, esto se traduce a que Bob tendría que verificar los SGT de los usuarios uno a uno secuencialmente para mantener sincronía con el contador, lo cual es impráctico en servidores con muchos usuarios haciendo *login* a la vez.

#### 8.4.1.4. Fake Ticket

En Caronte existe un tipo de *ticket* especial denominado el *Fake Ticket* (*Ticket Falso*), consistente en un TGT inválido generado por Caronte como respuesta a un identificador de usuario inexistente.

Este tipo de *tickets* tienen la intención de remediar posibles ataques de fuerza bruta inversa, en la que un atacante parte de contraseñas conocidas para intentar averiguar el email.

Es de importancia proteger el email del usuario tanto como su contraseña, con lo que un atacante nunca debe saber si el fallo en la autenticación ha sido por email o contraseña incorrectos.

Por esto Caronte debe generar una respuesta idéntica a la que generaría ante un usuario real. Para ello, se genera un IV del usuario y un TGT con información basura.

Por lo general, el IV del usuario nunca cambia (salvo por cambio de contraseña), con lo que el IV falsificado debe ser siempre el mismo para el mismo ID incorrecto (de lo contrario un atacante puede identificar un *ticket falso* simplemente observando si el IV cambia entre llamadas), esto se puede resolver usando una función *hash*.

```
email_hash = security.generate128Hash(params["email"])
```

El TGT se creará con la misma estructura que uno real, pero la información que contiene es incorrecta y no se corresponde con ningún ticket real.

```
fake_ticket_data = {  
    "name" : security.randB64(),  
    "version" : -1,  
    "ticket" : ""  
}  
for i in range(0, 5): fake_ticket_data["ticket"] += security.randB64()
```

Para evitar que el atacante pueda detectar que el *ticket* es falso, debemos impedir que pueda saber que el IV ha sido generado de un hash del ID que ha enviado, y que pueda leer la información del TGT. Para ello cifraremos el IV y el TGT con una clave interna de Caronte reservada exclusivamente para crear tickets falsos.

```
fake_hash = security.encryptKey(CARONTE_FAKE_TICKET_KEY, params["email"], email_hash)
```

```
fake_ticket = security.encryptKey(CARONTE_FAKE_TICKET_KEY, json.dumps(fake_ticket_data), ticket_iv)
```

### 8.4.2. Interfaz Criptográfica

Para el correcto funcionamiento del protocolo, tanto clientes como el servidor Caronte deben realizar exactamente las mismas operaciones criptográficas, usando los mismos algoritmos con las mismas entradas y salidas. No solo eso, sino que también debemos garantizar el funcionamiento del protocolo independientemente del lenguaje y la librería criptográfica que estemos usando. También se debe reforzar el uso de algoritmos criptográficos seguros. Si un algoritmo muestra ser vulnerable, debemos ser capaces de actualizarlo sin comprometer la lógica interna de los usuarios del protocolo. Es por eso que Caronte define una interfaz criptográfica que sirve principalmente de wrapper para enmascarar las operaciones y librerías criptográficas que estemos usando.

Esta interfaz se conoce como CaronteSecurity y existe una versión portada para cada uno de los lenguajes principales propuestos en el proyecto (C, Java, Python y JavaScript), haciendo uso de las librerías disponibles en cada lenguaje, pero manteniendo la coherencia en las operaciones, sus entradas y sus salidas.

Las principales operaciones que se definen en la interfaz se muestran en la tabla 8.3.

| Operación | Significado   | Entrada   | Salida  |
|-----------|---|---|---|
| RandB64   | Devuelve una cadena de bytes aleatorios de tamaño específico, codificado en Base64  | Byte count: número de bytes aleatorios a generar. | Cadena codificada en Base64 conteniendo N bytes aleatorios. |
| ToB64     | Convierte una cadena de bytes a una cadena codificada en Base64.  | Cadena de bytes.                                  | Cadena de caracteres en Base64.                             |
| FromB64   | Convierte una cadena de caracteres en Base64 a una cadena de bytes.   | Cadena de caracteres codificados en Base64.       | Cadena de bytes decodificada.                               |
| Pad       | Añade una serie de bytes de padding a una cadena de bytes para hacer que su tamaño sea múltiplo del tamaño de bloques del algoritmo de cifrado (16 bytes para AES). | Cadena de bytes o caracteres sin padding.         | Cadena de bytes con padding.                                |
| Unpad     | Elimina el padding de una cadena de bytes.  | Cadena de bytes con padding.                      | Cadena de bytes sin padding.                                |

Tabla 8.3.1

| <b>Operación</b>  | <b>Significado</b>  | <b>Entrada</b>   | <b>Salida</b>   |
|-------------------|---|--|---|
| Generate128Hash   | Genera un hash de la entrada de 128 bits de tamaño.   | Cadena de bytes sobre la que hacer hashing.  | Resultado de la función hash.   |
| Generate256Hash   | Genera un hash de 256 bits de la entrada.   | Cadena de bytes sobre la que hacer hashing.  | Resultado de la función hash.   |
| EncryptKey        | Genera un texto cifrado a partir de un texto plano y una clave criptográfica de 256 bits.   | Texto en plano a cifrar. Clave de 256 bits. IV de 128 bits.  | Texto cifrado y codificado en Base64.   |
| DecryptKey        | Genera un texto plano a partir del texto cifrado, la clave y el IV usados para cifrar.  | Texto a descifrar. Clave de 256 bits. IV de 128 bits.  | Texto descifrado.   |
| EncryptPBE        | Igual que EncryptKey pero usando una contraseña. El principal objetivo de esta función es el de derivar la contraseña en una clave de 256 bits para delegar a EncryptKey. | Texto en plano a cifrar. Contraseña de cifrado. IV de 128 bits.                                      | Texto cifrado y codificado en Base64.   |
| DecryptPBE        | Igual que DecryptKey pero usando una contraseña. El principal objetivo de esta función es el de derivar la contraseña en una clave de 256 bits para delegar a DecryptKey. | Texto a descifrar. Contraseña de cifrado. IV de 128 bits.  | Texto descifrado.   |
| DeriveText        | Dado un texto plano, un IV y un número de iteraciones, la función devuelve un texto derivado del original.  | Texto a derivar. IV a usar en el cifrado del texto derivado. Numero de iteraciones en la derivación. | Devuelve un texto derivado del original y (opcionalmente) una clave derivada del texto usada para cifrar el propio texto. |
| VerifyDerivedText | Verifica que un texto derivado se corresponde con un texto plano.   | Texto derivado. Texto plano. IV y número de iteraciones usados para derivar el texto.                | Resultado booleano.   |

Tabla 8.3.2



### 8.4.3. Almacenamiento de credenciales

Uno de los principales problemas en la autenticación de usuarios es el almacenamiento de las credenciales. No solo es importante evitar transmitir las credenciales por la red, sino también proteger aquellas credenciales almacenadas en la Base de Datos.

Para entender cómo Kerberos resuelve el problema debemos primero estudiar las diversas soluciones que se han ideado y propuesto hasta ahora, así como sus ventajas y debilidades.

(1) La primera solución es el uso de una función *hash* para almacenar un resumen unidireccional de la contraseña. Esta es la solución que usa Kerberos.

Las funciones *hash* tienen una serie de características que las hacen ideales:

- La salida es unidireccional; no puedes calcular la contraseña original conociendo el *hash*.

Sin embargo hay problemas con el uso de funciones *hash*:

- Colisiones: dependiendo del algoritmo un atacante puede inferir varias contraseñas que generan el mismo *hash*.

- Velocidad: las funciones *hash* son muy rápidas, lo que facilita a un atacante inferir la contraseña original mediante fuerza bruta.

- La salida es de un tamaño fijo y dictado por el algoritmo *hash*, no por el de cifrado.

(2) Para resolver el problema con las colisiones, la otra técnica que se usa es el cifrado de las contraseñas mediante una clave maestra conocida por el servidor.

Sin embargo esta solución tiene una serie de inconvenientes:

- Si el atacante roba la clave maestra, todas las contraseñas de los usuarios quedan comprometidas. Toda la seguridad del sistema depende de una sola clave.

- No resuelve el problema de la velocidad, de hecho los algoritmos de cifrado simétrico son bastante eficientes temporalmente.

(3) El problema de la velocidad se resuelve mediante el uso de una KDF, almacenando la clave criptográfica derivada de la contraseña y del tamaño dictado por el algoritmo de cifrado. Algunos ejemplos de KDF son BPKDF2, bcrypt o scrypt.

Esta es la técnica más aceptada y recomendada. Sin embargo los datos que almacenamos siguen siendo un resumen y su tamaño sigue siendo dictado por el algoritmo de cifrado. Es necesario poder usar un algoritmo que nos permita tener las ventajas que ofrece cada solución: evitar colisiones, ralentizar el proceso y generar un “hash” que podamos almacenar en lugar de la contraseña, pero sin perder información sobre la contraseña.

(4) Una técnica conocida que combina ambas soluciones es el *Ad-Hoc Hashing*, consistente en cifrar la contraseña usando una clave criptográfica derivada de esta misma, resolviendo el problema de las colisiones inherentes a las funciones *hash* (ya que la salida contiene los datos de entrada en sí en lugar de un resumen) pero manteniendo la unidireccionalidad (no puedes descifrar la contraseña cifrada si no conoces la propia contraseña).

Sin embargo para que esta solución sea completa y resuelva otros problemas como el de la velocidad, debemos hacer uso de una KDF para derivar la contraseña en la clave criptográfica.

A esta operación Caronte la denomina *Función de Derivación de Texto*. El siguiente pseudocódigo la refleja:

```
def deriveText(text, salt, iters):  
    return encrypt(plaintext=text, key=KeyDerivationFunction(text, iters), IV=salt)
```

Los parámetros de entrada de esta función son:

- Text: el texto en plano a derivar. De cualquier tamaño.
- Salt: número que se usará como IV en el cifrado, preferentemente aleatorio.
- Iters: número de iteraciones a realizar por la Key Derivation Function.

La salida es un texto derivado de tamaño igual o superior al original (¡nunca inferior!). Usando el texto plano y el número de iteraciones, se genera (mediante una KDF) una clave criptográfica que se usará, junto al IV, para cifrar el texto plano.

El resultado es una salida similar a un *hash* que se puede usar para sustituir el texto original, conteniendo toda la información del texto original en si pero inaccesible si no se conoce el propio texto original. Si el IV (o salt) y las iteraciones son estáticas (por ejemplo un *hash* del texto original), la salida también será estática; esto es útil para generar siempre el mismo texto derivado. Por el contrario, un IV aleatorio genera un texto derivado aleatorio.

#### 8.4.4. Base de Datos del servidor

El proyecto software presentado hace uso de las primitivas propias de Django para la construcción de la Base de Datos y los DAOs.

Las tablas definidas, así como su contenido, significado y funcionalidad, se explicará a continuación.

La base de datos nos debe permitir almacenar la siguiente información:

- Los datos de un usuario: identificador (email), identificador derivado (ID), contraseña derivada (¡nunca la original!), IV usando como *salt* en el proceso de derivación de la contraseña y *ticket* actualmente en uso (si lo hay).
- Los tickets que han sido generados por los usuarios: un *token* (número aleatorio) que identifica el ticket, una clave temporal con la que el usuario cifrará sus *tickets*, el contador actual del *ticket* y el usuario propietario del *ticket*.
- La sesión entre dos usuarios: referencia a los *tickets* de Alice y Bob en el momento de establecer la sesión, la clave de sesión y el IV usado para cifrar la clave de sesión.

Es importante poder monitorizar la actividad de los usuarios del sistema. Es por ello que se almacenan todos los *tickets* que ha generado un usuario, no sólo el que está actualmente en uso. Debe ser posible poder observar el historial de *tickets* y el usuario que los ha generado. El contador del *ticket* también nos da la información de el número de veces que se ha usado ese mismo *ticket*.

La sesión entre dos usuarios debe poder identificar a los tickets que han sido usados para la sesión, lo cual también nos da información de los usuarios en si pues se puede inferir a partir del *ticket*.

Las tablas de la BBDD tienen también *timestamps* del momento de su creación, lo que nos permite saber cuando se ha registrado un usuario, cuando se ha generado un nuevo ticket y cuando se ha establecido una sesión. Otro *timestamp* se encuentra en el *ticket* para saber cuando ha sido la última vez que se usó.

La figura 5.3 del apartado 5.4 muestra el diagrama Entidad-Relación de la Base de Datos.

#### 8.4.4.1. Tabla del Usuario

La información almacenada sobre el usuario debe ser la mínima posible, necesario para hacer una identificación, pero sin reflejar detalles críticos, en especial la contraseña.

Los campos que contiene la tabla de usuario se detallan en la tabla 8.4.<sup>4</sup>

| Nombre        | Tipo        | Atributos                         | Descripción  |
|---------------|-------------|-----------------------------------|--|
| Name          | VARCHAR     | NOT NULL,<br>NOT BLANK            | Nombre del usuario.  |
| Email         | VARCHAR     | NOT NULL,<br>NOT BLANK,<br>UNIQUE | Identificador principal del usuario.   |
| Email Hash    | VARCHAR     | NOT NULL,<br>NOT BLANK,<br>UNIQUE | Hash (o texto derivado) del identificador del usuario.   |
| Password      | VARCHAR     | NOT NULL,<br>NOT BLANK,<br>UNIQUE | Texto derivado de la contraseña del usuario.   |
| IV            | VARCHAR     | NOT NULL,<br>NOT BLANK            | Vector de Inicialización usado para derivar la contraseña del usuario. Debe ser aleatorio. Se almacena tanto en texto plano como cifrado, usando una contraseña derivada estática. |
| Active Ticket | FOREIGN KEY | ON DELETE SET NULL                | Ticket actualmente activo en el usuario. Puede no haber.   |
| Joined        | DATETIME    | NOT NULL,<br>AUTO NOW ADD         | Marca temporal de la creación del usuario.   |
| Last Active   | DATETIME    | NOT NULL,<br>AUTO NOW             | Marca temporal de la última actividad del usuario.   |

Tabla 8.4

4 Para esta y demás tablas de la BBDD, se omite el campo ID (clave primaria de la tabla), pues es Django el que configura y administra este campo.

#### 8.4.4.2. Tabla del Ticket

La base de datos almacena también toda la información referente al *ticket*, su creación, uso, y demás datos importantes para el protocolo. Ver tabla 8.5 a continuación para más detalles.

| Nombre    | Tipo           | Atributos                         | Descripción   |
|-----------|----------------|-----------------------------------|---|
| TimeStamp | DATETIME       | NOT NULL,<br>AUTO NOW ADD         | Marca temporal de la creación del <i>ticket</i> .   |
| Temp Key  | VARCHAR        | NOT NULL,<br>NOT BLANK            | Clave criptográfica aleatoria de 256 bits codificada en Base64. Usada para cifrar SGTs.   |
| Token     | VARCHAR        | NOT NULL,<br>NOT BLANK            | Número identificador del <i>ticket</i> con respecto a su dueño. El procedimiento para generar el <i>Token</i> así como su significado no están definidos de manera explícita en el protocolo.<br>El proyecto de ejemplo genera un número aleatorio. |
| Counter   | INTEGER        | DEFAULT=0                         | Contador del <i>ticket</i> . Debe estar sincronizado con el usuario para evitar ataques <i>replay</i> . Sirve también para saber el número de veces que se ha usado el <i>ticket</i> .  |
| Valid     | BOOLEAN        | DEFAULT=True                      | Determina la validez del <i>ticket</i> .  |
| Owner     | FOREIGN<br>KEY | NOT NULL,<br>ON DELETE<br>CASCADE | Propietario del <i>ticket</i> .   |

Tabla 8.5

#### 8.4.4.3. *Tabla de la Sesión*

La siguiente tabla de la base de datos define a modo de registro cada sesión establecida entre dos usuarios mediante un KGT. Ver tabla 7.6 a continuación para más detalles.

| Nombre    | Tipo        | Atributos                 | Descripción  |
|-----------|-------------|---------------------------|--|
| TimeStamp | DATETIME    | NOT NULL,<br>AUTO NOW ADD | Marca temporal del establecimiento de la sesión.       |
| Ticket A  | FOREIGN KEY |                           | <i>Ticket</i> del usuario A en el inicio de la sesión. |
| Ticket B  | FOREIGN KEY |                           | <i>Ticket</i> del usuario B en el inicio de la sesión. |
| Key       | VARCHAR     | NOT NULL                  | Clave de sesión de 256 bits en Base64.                 |
| Key IV    | VARCHAR     | NOT NULL                  | IV usado para cifrar la clave de sesión.               |

*Tabla 8.6*

#### **8.4.5. API REST del Servidor**

La comunicación entre usuarios y el servidor Caronte se realiza mediante una API RESTful, compuesta de dos rutas principales correspondiente a las dos etapas del protocolo: una para autenticar usuarios, y otra para verificar tickets y establecer una sesión.

Cada ruta se compone a su vez de los cuatro métodos HTTP; GET, POST, PUT y DELETE. Dependiendo de la ruta, tendrá un comportamiento, input y output específicos.

Todos los parámetros de entrada y de salida se representan mediante JSON.

#### 8.4.5.1. Caronte Authenticator

La funcionalidad de esta ruta está enlazada a la comunicación entre Caronte y un usuario.

| Método | Parámetros                                | Salida                  | Descripción   |
|--------|---|-------------------------|---|
| GET    | -   | Status, msg, params     | Devuelve el nombre y versión de Caronte, así como la configuración criptográfica usada por el servidor (algoritmos criptográficos, tamaño de las claves, número de iteraciones para la KDF, etc). |
| POST   | Identificador de usuario (email derivado) | Status, IV, TGT, tgt_iv | Devuelve un nuevo TGT para el usuario. Si el usuario no existe el resultado será un <i>ticket falso</i> .   |
| PUT    | SGT                                       | Status, user            | Devuelve información sobre el usuario. También sirve para activar el SGT.   |
| DELETE | SGT                                       | Status, msg             | Realiza un log-out. Se inhabilita el ticket actual del usuario.   |

Tabla 8.7

Se puede acceder a esta ruta mediante la URL /crauth/.



#### 8.4.5.2. Ticket Validator

La funcionalidad principal de esta ruta es validar los tickets del usuario y, en caso de los KGT, establecer una sesión. La URL es /validate/ y sólo acepta llamadas al método POST, que cambiará su funcionalidad según los parámetros enviados.

| Método | Parámetros | Salida                              | Descripción  |
|--------|------------|-------------------------------------|--|
| POST   | SGT        | Status, msg                         | Recibe un SGT de un usuario y lo verifica. Sirve para activar el <i>ticket</i> si es la primera vez que se usa.  |
| POST   | KGT        | Status, msg, key, key_other, key_iv | Recibe un KGT que contiene el ID del proveedor y el SGT del usuario cifrado con la clave del proveedor. Si el KGT valida correctamente se genera una clave cifrada para el proveedor y el usuario. |

Tabla 8.8

### 8.4.5.3. Ejemplo de registro y proveedor

El registro de usuario y credenciales no es una tarea contemplada por el protocolo Caronte. Se recomienda el uso de herramientas externas especializadas.

Sin embargo, a modo de demostración para la página web de ejemplo se proporciona una ruta en la URL /register/ (tabla 8.9) que permite registrar usuarios mediante el uso de una clave maestra o cambiar las credenciales de un usuario registrado (mediante el campo “extra\_data” de los SGT).

| Método | Parámetros | Salida      | Descripción   |
|--------|------------|-------------|---|
| POST   | User, IV   | Status, msg | Recibe los datos del usuario cifrados con la clave maestra y un IV aleatorio. Se registra el usuario en el sistema. |
| PUT    | SGT        | Status, msg | Recibe la contraseña antigua y la nueva dentro del SGT y actualiza el usuario.                                      |

Tabla 8.9

Adicionalmente, Caronte proporciona un ejemplo de proveedor de servicios mediante la API REST en la URL /provider/, la cual contiene los siguientes métodos (tabla 8.10):

| Método | Parámetros | Salida           | Descripción   |
|--------|------------|------------------|---|
| GET    |            | Datos a consumir | Obtiene los datos a consumir por el usuario. Debe haber establecida una clave de sesión que se usará para cifrar los datos. |
| POST   | Ticket     | Clave de sesión  | Se realiza la operación de <i>login</i> . El usuario proporciona su SGT y el proveedor retorna la clave de sesión cifrada.  |

Tabla 8.10

El procedimiento para autenticar un usuario y establecer una sesión desde el punto de vista del proveedor se puede realizar con el siguiente pseudocódigo.

Suponemos que el proveedor ha realizado una conexión con Caronte.

```
# connect to Caronte Server
car_cli = client.CaronteClient("localhost", request.META['SERVER_PORT'])

# login to Caronte
if not car_cli.login("sample.provider@caronte.com", "Sampl3Pr0videR"):
    log("Could not connect to caronte server")
    return invalidData()
```

El proveedor transforma el SGT del usuario en un KGT y lo verifica con Caronte para establecer una sesión.

```
# validate other ticket and establish a session
if not car_cli.validateTicket(params["ticket"], True):
    log("Could not verify user ticket %s\n"%(params["ticket"]["ID"]));
    return invalidData()
```

Si el *ticket* valida, el proveedor procede a obtener la clave de sesión para el usuario.

```
# obtain the session key for the user
other_key = car_cli.getOtherKey(user_id)
```

Y procede a retornar la clave de sesión cifrada al usuario.

```
# respond to the user with the session key
res = {
    "status" : STAT_OK,
    "msg" : "Tickets verified",
    "key" : other_key
}
return JsonResponse(res)
```

#### 8.4.6. Código del cliente

El proyecto proporciona una librería para diversos lenguajes que implementa la lógica del funcionamiento del cliente. La tabla (8.11) explica los métodos disponibles.

| Nombre         | Parámetros  | Salida                                    | Descripción   |
|----------------|---|---|---|
| Login          | Email, password   | Boolean                                   | Realiza el handshake con Caronte para crear los SGT.  |
| GetTicket      | extra_data (opcional)                                   | SGT válido, ID del usuario, IV de cifrado | Genera el próximo SGT válido para autenticarse. Esta operación incrementa el contador del ticket, si el SGT no se consume (no se envía a Caronte) se perderá la sincronía y será necesario revalidar el ticket.       |
| GetUserDetails | Update (boolean, por defecto False)                     | Detalles sobre el usuario.                | Si update es True (o no hay datos locales sobre el usuario) se enviará un SGT a Caronte para obtener detalles sobre el usuario (email, nombre y fecha de registro).   |
| Logout         |   | Boolean                                   | Realiza un logout con Caronte. Los tickets de esta sesión quedarán inválidos.   |
| UpdateUser     | Name, old password, new password                        | Boolean                                   | Actualiza la contraseña del usuario mediante SGT.   |
| ValidateTicket | Other (SGT, opcional), Session (boolean, default=false) | Boolean                                   | Para verificar un SGT (el propio si no se proporciona uno). Si el parámetro <i>session</i> es <i>True</i> (y el <i>ticket</i> a verificar no es el propio), se convierte el SGT en un KGT para establecer una sesión. |

Tabla 8.11.1

| Nombre           | Parámetros        | Salida  | Descripción   |
|------------------|-------------------|---|---|
| RevalidateTicket |                   | Boolean   | Vuelve a realizar el <i>handshake</i> con Caronte para generar un nuevo <i>ticket</i> .   |
| InvalidateTicket |                   | Boolean   | Envía un <i>ticket</i> inválido a Caronte. Siempre devuelve <i>False</i> . Se necesita hacer <i>revalidateTicket</i> para volver a activar la sesión.                             |
| SetOtherKey      | Key               | El ID del usuario con el que se establece una sesión. | Establece una sesión con un nuevo usuario. El parámetro debe ser lo que devuelve Caronte tras verificar el KGT. Retorna el ID del usuario con el que se ha establecido la sesión. |
| GetOtherKey      | other_email       | String  | Clave del otro usuario según la creó Caronte.   |
| EncryptOther     | other_email, data | Ciphertext  | Cifra un texto plano usando la clave establecida con el usuario proporcionado.  |
| DecryptOther     | other_email, data | Plaintext   | Descifra un texto usando la clave establecida con el usuario proporcionado.   |

Tabla 8.11.2

#### 8.4.6.1. Login y generación de tickets

El proceso de *login* se hace mediante un *handshake* a dos vías.

El usuario realiza una petición de comunicación mediante su ID (email derivado).

```
params = {"email": CaronteSecurity.deriveText(email, CaronteSecurity.generate128Hash(email))[0]};  
self.conn.request("POST", CaronteClient.CR_LOGIN_PATH, body=json.dumps(params))
```

Para descifrar el TGT el usuario debe derivar la contraseña usando los datos recibidos por Caronte.

Primero derivamos una contraseña estática:

```
self.p1 = CaronteSecurity.deriveText(password, CaronteSecurity.generate128Hash(password), data["pw_iters"]);
```

Usamos la contraseña derivada estática para descifrar el IV:

```
IV = CaronteSecurity.toB64(CaronteSecurity.decryptPBE(self.p1, data["IV"], CaronteSecurity.generate128Hash(self.p1)))
```

Derivamos la contraseña otra vez, ahora usando el IV descifrado:

```
self.p2 = CaronteSecurity.deriveText(password, IV, data["pw_iters"]);
```

El usuario procede a usar esta contraseña derivada para descifrar el TGT:

```
plain_ticket = json.loads(CaronteSecurity.decryptPBE(self.p2, data["TGT"], data["tgt_iv"]));
```

El usuario procede a extraer la información del TGT descifrado y construye el SGT:

```
self.ticket = {"t":plain_ticket["token"], "c":1, "user_iv":data["IV"], "email":email};  
self.ticket_key = plain_ticket["tmp_key"]
```

La información del *ticket* se cifra usando la clave temporal encontrada dentro del TGT:

```
valid_ticket = CaronteSecurity.encryptKey(self.ticket_key, json.dumps(ticket_data), ticket_iv);
```

Ahora podemos enviar nuestro primer SGT a Caronte para validar, mediante la API de *Validator* o una petición de datos del usuario (método PUT de la ruta *crauth*).

#### 8.4.6.2. Validación de otros tickets

La validación un propio *ticket* se realiza enviando el SGT cifrado junto al identificador de usuario. Partiendo del código anterior, la siguiente información se enviaría al validador de *tickets* de Caronte:

```
{"ID":self.ticket["user_iv"], "IV":ticket_iv, "SGT":valid_ticket};
```

Algo muy diferente es validar el *ticket* de otro usuario para establecer una sesión. Se enviará el ID de nuestro usuario, pero el *ticket* enviado será el SGT del otro usuario cifrado usando la clave temporal de nuestro *ticket*:

```
params["ticket"] = {  
    "ID":self.ticket["user_iv"],  
    "IV":ticket_iv,  
    "KGT":CaronteSecurity.encryptKey(self.ticket_key, other_ticket, ticket_iv)  
}
```

Hecho esto se procede a comunicar los datos a Caronte para validarlos:

```
self.conn.request("POST", CaronteClient.VALIDATE_PATH, headers=self.header, body=json.dumps(params))
```

Si todo verifica bien, Caronte nos devolverá la clave de sesión y el IV para descifrarla:

```
tmp_key = json.loads(CaronteSecurity.decryptKey(self.ticket_key, data["tmp_key"], data["tmp_iv"]))
```

Procedemos a enviarle al usuario su clave de sesión (*tmp\_key\_other*). La librería proporcionada almacena la clave de sesión e información del usuario conectado en un diccionario:

```
self.valid_users[tmp_key["ID_B"]] = {  
    "key":tmp_key["key"],  
    "key_other":data["tmp_key_other"],  
    "iv":data["tmp_iv"],  
    "email":tmp_key["email_B"]  
}
```

La clave de sesión viene cifrada junto con datos sobre el usuario que se conecta (ID, email). El usuario que recibe la otra clave debe descifrarla igual que lo ha hecho el proveedor de servicios:

```
tmp_key = json.loads(CaronteSecurity.decryptKey(self.ticket_key, info["key"], info["iv"]))  
self.valid_users[tmp_key["ID_A"]] = {  
    "key": tmp_key["key"],  
    "iv": info["iv"],  
    "key_other": None,  
    "email":tmp_key["email_A"]  
}
```

## 9. Conclusión y Bibliografía

La ciberseguridad es uno de los aspectos más interesantes de la ingeniería informática y uno de los que está en mayor desarrollo en la actualidad. La lucha entre hackers y desarrolladores es constante y tediosa, y parece no tener fin.

Cuanto mejor son las herramientas y medidas de seguridad, mejor son las herramientas para abatirlas.

Ningún sistema es 100% seguro con total certeza y el factor humano entra mucho en juego. Sin embargo, toda medida que tomemos para contrarrestar y ralentizar los ataques son siempre de suma importancia. En cuanto a Caronte, todavía queda mucho que mejorar y testar, pero la propuesta es una prometedora solución al problema de autenticación de usuarios en un tráfico red poco fiable.

A continuación una serie de recursos online que han servido de base para este proyecto y pueden servir para ampliar el conocimiento sobre la ciberseguridad.

\* La mayor fuente de información y conocimientos para este proyecto proviene de la *Open Web Application Security Project* (OWASP).

- Su página web oficial:

[https://www.owasp.org/index.php/Main\\_Page](https://www.owasp.org/index.php/Main_Page)

- Libro sobre las 10 principales amenazas cibernéticas:

<https://www.owasp.org/images/5/5e/OWASP-Top-10-2017-es.pdf>

- Diferentes tipos de ataques por categoría:

<https://www.owasp.org/index.php/Category:Attack>

\* Explicación de los timestamps de Kerberos y su uso para evitar ataques replay.

<https://www.itprotoday.com/security/understanding-how-kerberos-authentication-protects-against-replay-attacks>

\* Sobre los tickets de Kerberos y vulnerabilidades.

<https://www.tarlogic.com/blog/tickets-de-kerberos-explotacion/>

\* Sobre password hashing: los DOs y DON'Ts

<https://crackstation.net/hashing-security.htm>

\* Más sobre algoritmos para cifrar contraseñas

[https://www.ibm.com/support/knowledgecenter/en/SSVJJU\\_6.4.0/com.ibm.IBMDS.doc\\_6.4/ds\\_ag\\_srv\\_adm\\_pwd\\_encryption.html](https://www.ibm.com/support/knowledgecenter/en/SSVJJU_6.4.0/com.ibm.IBMDS.doc_6.4/ds_ag_srv_adm_pwd_encryption.html)

<https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/>