Universidad de Murcia **Facultad de Informática**

Integración de Kerberos con servicios Cloud (OpenStack)

Trabajo Fin de Grado

Autor:
Víctor Manuel Ruiz Sánchez
<victormanuel.ruiz@um.es>

Tutores:
Gabriel López Millán
<gabilm@um.es>
Rafael Marín López
<rafa@um.es>

21 de julio de 2014

Resumen

El uso de Internet por parte de las aplicaciones que utilizamos diariamente es cada vez más frecuente. Junto a este hecho, destaca el crecimiento que está experimentado la denominada Computación en la Nube, que se basa en el servicio de plataformas y aplicaciones software alojadas en remoto y accedidas a través de la red. El proyecto Open Source OpenStack, apoyado por grandes multinacionales, es un referente entre el software destinado al despliegue de estos entornos.

En base a ello, y como manera de colaboración para la mejora del proyecto OpenStack, este trabajo consiste en el análisis, desarrollo y prueba de un mecanismo de autenticación basado en Kerberos para dicha plataforma de Cloud Computing, que en última instancia permitirá a organizaciones que ya tienen desplegadas arquitecturas Kerberos su integración con este nuevo paradigma. Además estas podrán extender a Openstack la propiedad de Single Sing On (SSO) que Kerberos ya aporta a servicios como SSH y FTP.

Índice

Extended Abstract

For the last years, Internet has become the core of modern applications. This is the case of happening in all kind of software, from instant messaging applications such as Whatsapp [1], to streaming media players such as Spotify [2] or team collaboration tools such as Google Docs [3]. All of them have conquered the market share that, until now, had their offline counterparts. Besides, the Cloud Computing [4] model has also appeared. This paradigm is based on service's offer, in form of software or platforms for development, through public or private networks, by using of virtualization and high availability techniques. As an example of applications using this model, Dropbox [5], Gmail [6] and Amazon Web Service (AWS) [7] can be cited.

The Cloud gives tremendous benefits to their users [8], such as On-demand self-services - where the consumer can improve his computing resources, like server time or network storage, as needed without human interaction been required -, and the high availability provided by the use of virtualization techniques which reduces the possibility of a shutdown in the service's uptime. In business Cloud - such as AWS -, new advantages can be found. Some examples are the service cost reduction given by the high business's volume, the capability of bringing the data from different data centers across the world to minimize latency in connections, and the investment in risk management which could not be afforded by lower-size companies.

Nevertheless, there are important security threats that must be kept in mind [8]. For example, the consumer loses control over their data and it could lead to do not be able to ensure that the data is processed according to the local laws. Also, security exposures in the cloud's archiquecture are out of the consumer scope and the service's politics may ban the realization of security analysis in applications running inside them. Furthermore, other classic risks on the services access through a non-secure network are here present: information could be eavesdropped or modified while is in transit over the network, user's session may be hijacked or the information could be accessed by people without authorization.

Three types of Cloud architectures can be described [4]:

- The first model is the 'Infrastructure as a Service' (IaaS). This kind of Cloud allows offering computing time in virtual machines; creating virtual networks which interconnected these machines, and managing virtual storage devices built over physical disk RAIDs. IaaS is the lowest abstract level in the models of Clouds.

- The second type is the 'Platform as a Service' (PaaS). It gives to the consumer the capability to deploy workspaces with tools such a libraries, services, and IDEs to develop their applications, but relieving him of the underlaying details like network or operating systems.

- The third and highest abstract level is the one named 'Software as a Services' (SaaS), witch allow to run applications over the Cloud. Generally, web technologies are used to offer these services.

In spite of the flexibility given by the IaaS Clouds, the most usual configuration nowadays is PaaS Cloud with services like Dropbox, Gmail, Google Doc, Spotify as examples. Even though the fame of the SaaS model, there is also a lot of interest in the development of the IaaS Clouds. Because of that, and with the idea of give to any organization the capacity to deploy their own PaaS Clouds in mind, the OpenStack project [9] was created. This Open Source project contains a complete collection of modules to launch Infrastructure as a Service Cloud Computing environments.

Because of its free nature and numerous functionalities, OpenStack has become one of the most important references among the software used to create these environments. It has caused that companies such as HP, Cisco or Intel support it through different ways like financial support, collaboration between developers and publicity. OpenStack is not a monolithic system. It consists of different modules designed to do a specify group of tasks -such as network control, CPU virtualization, or disk storage-. In this work, the most interesting module is the Identity Management, called Keystone. It is used by the rest of the modules to give them authentication and high-level authorization over users of the OpenStack enviroment.

If we compared the security related threats listed before, with the current state of the OpenStack project, we would see that these hazards have been controlled. Keystone allows clients to use secure mechanisms such as HTTPs to protected the integrity and privacy of the communication, or OAuth1.a [10] as a way to authenticate users. Also, there is a fine-grained resource control access by the establishment of user roles, projects - called tenants -, and domains. In spite of the fact that, these technologies could cover almost all the basic use cases, there is yet a lot of work to do.

A concrete example is the absence of an authentication mechanism based in the Kerberos [11]. This have caused that organizations like CERN [12] have considered implement their own solutions to integrate their Kerberos deployment with the OpenStack Cloud [13]. The reason why Kerberos is so important is that, is a world-wide recognised secure

mechanism which allows to authenticate users using symmetric cryptography by the collaboration of a reliable third-party. At present, it's integrated with application services as SSH [14], FTP [15] or SMTP [16]. In addition to that, Kerberos allows the access to several services without the need of repeating the authentication process - as long as the session stayed alive, by default for twenty four hours -. It is knowns as Single Sign On. Besides the above, the use of a technique from the Kerberos architecture called Cross-Realm [17], let users to use the services from external organizations.

The use of a non-public service by a external user is known as Identity Federations. To make it possible, The user needs to be enrolled with an Identity Provider (IdP) - examples of Identity Providers are Universities, Research Centers, Governments or even companies such as Google and Facebook -. The IdP is in charge of the user's information storage. The user can then access to any service offered by the IdP's organization - for example FTP servers -, but also to any services from organisations which have an a federation agreement with their IdP- as occur in the case of the eduroam [18] network connection -. The service provider has no clue of who the user may be, but it can discover where is its IdP and to delegate the authentication process to it. This feature implies a lot of benefits for the users, who don't need to give their information to multiple organizations to use their services; for the system administrators, who has seen how their work has been reduced due to they don't have to manage external users information in their databases; and even for the organizations itself, thanks to the business profit that could be archived by renting their services to others organizations.

How this delegation will take place depends on the used federated mechanism. As said above, Kerberos Cross-Realm is a way to get this federated access, but there are more techniques that relay in differents mechanisms to achieve this purpose. Based on the document 'Identity Federations Beyond the Web: A survey' [19] there are three main categories of identity federation technologies:

1. **Web-based Identity Federations**: It's is used to decoupled the end user's access to a web services from his authentication and authorization. The main benefit of this system is that the service provider does not need to worry about how the authentication and authorization take places, which allows focusing on a better service to the user. This model is based on standards or building web service identity federations such as SAML [20], OAuth2.0 [21], OpenID [22], etcetera...

2. **AAA-based Identity Federations**: network access service providers have also used the benefits of identity federation. In this case the user does not get access to a web service, but rather he will gain access to a network. The standards used in

this model are the Authentication, Authorization and Accounting protocols - such as RADIUS [23][24] and the Extensible Authentication Protocol (EAP) [25].

3. **Application-agnostic Identity Federations**: the architectures for federation cited above are very useful in some concrete scenarios, but it is not possible to extend their use to other services like cloud infrastructures, file store, e-mail, printers, databases or SSH connections. For this reason, researchers are working in the creation of a main standards able to extend the Identity Federation to any kind of application. Specifically, the IETF is developing an architecture called Application Bridging for Federated Access Beyond Web (ABFAB) [26].

   ABFAB appeared as the result of the project Moonshot developed by the Janet organization [27], which main goal is to obtain a single unifying technology for Application-agnostic Identity Federations. It is based in the use of mechanisms and protocols such as AAA architectures - with RADIUS as protocol -, EAP, GSS-API [28], or SAML.

Summing up the above, Kerberos is a great system for the authentication of users and services that have not been implemented yet into the Openstack environment. Beside the basic use of Kerberos, it also provides others advantages such as integration in the Kerberos Single Sign On environment, and the capability to get Identity Federation with Kerberos Cross-Realm. Because of all of this, there are reasons enough to establish the adaptation of Kerberos into the Keystone module as the goal of this project.

To get success in this goal, an authentication plugin for the Keystone module has been created and the Swift client software has also been modified to implement it. The plugin's work is divided in three different phases:

1. **Discovery phase**: the first step is to obtain and Ticket Grating Ticket from the Kerberos Authentication Server in the user's realm. This point was intentionally left out of the mechanism process because it is usual to be achieved along with the user device login. In other cases, it could be also obtained by using the Kerberos client command `kinit`. Then, it will check if the Kerberos mechanism is available for user authentication by sending and HTTP request to the Keystone service. If the mechanism is supported, the response will carries information about the Keystone principal - its identificator in Kerberos terminology -, in the KDC and an universally global identifier which will be used to group futures messages under the same conversation context. In other case, the response will contain a list with the authentication mechanisms available in this specified Keystone server.

2. **Negotiation phase**: using the information obtained in the previous phase, the client starts a GSS-API security context. While creating it, the GSS-API will obtain the Service Ticket from the Kerberos Ticket Granting Server in the user's realm. Then, this ticket is wrapped into a GSS-API token and will be pass through to the Keystone Server. In this side, the Keystone will also start a GSS-API security context. As a result, the Keystone will have authenticated the user and, in addition to that, it will obtain another GSS-TOKEN that must be send to the client. This GSS-API token will be used to also authenticate the service.

3. **Finalization phase**: once each side has been authenticated, the client will ask to obtain the Keystone token. With the help of the negotiation identifier, the server will recover the GSS-API security context and check that the user was correctly authenticated before. Finally, Keystone will server the Keystone token to the client.

Once the client has obtains the Keystone token, the user can access to the desired service by adding it to the headers of the service request message.

Furthermore, along with the design, implementation and test of this mechanism, a theoretical study about the possibility of using this process to obtaining identity federation authentication with Kerberos Cross-Realm has been made. The conclusions of this study are that, in principle it seems possible to obtain without modifications be required. Nonetheless, it should be checked in a test scenario to known with certainty.

Finally, a brief proof of concept on how to integrate the Kerberos mechanism in the ABFAB/Moonshot environment, developed by the Kent's University for the Openstack project, has been made. Although it has been made a complete authentication with Kerberos, it was not in a federated way as it would be expected from something integrated with Moonshot. Future work is required in this topic.

1. Introducción

Internet se ha convertido en el núcleo fundamental en torno al cual se sustentan las aplicaciones modernas, tal y como hemos ido experimentando en las últimos años. Muestra de ello es el auge de las aplicaciones de mensajería tipo Whatsapp [1], de contenido multimedia por streaming como Spotify [2] o herramientas de trabajo colaborativas como Google Docs [3], que han conseguido desbancar, en muchas ocasiones, a sus homólogos *offline*. Por otra parte y acompañando a la gran proliferación de las comunicaciones en red actual, ha surgido el concepto de Computación en la Nube (*Cloud Computing*) [4]. Este paradigma se constituye como una oferta de servicios, en forma de software o plataformas de desarrollo, a través de redes públicas o privadas, utilizando para ello técnicas de virtualización y alta disponibilidad. Algunos ejemplos de aplicaciones que usan este paradigma son Dropbox [5], Gmail [6] y Amazon Web Service [7] (AWS en adelante).

La Nube ha supuesto grandes beneficios para sus usuarios [8] como el escalado rápido de los recursos conforme aumenta la necesidad, sin tener que existir necesariamente mediación humana, y la alta disponibilidad de los servicios gracias al uso de técnicas de virtualización. En Nubes destinadas al negocio, como AWS, aparecen otras ventajas, entre las que destacan la reducción de los costes de los servicios gracias a su oferta en gran escala, la capacidad de brindar el servicio desde diferentes Centros de Procesamiento de Datos (CPDs) según la saturación de la red, o las inversiones en gestión de riesgos, cuyo gasto puede no ser asumible por empresas con menor volumen de beneficios.

No obstante, también aparecen una serie de riesgos en la seguridad que es necesario considerar [8]: se produce una pérdida del control sobre los datos que puede implicar la no garantía de que la información sea tratada de acuerdo a las leyes vigentes; las vulnerabilidades en la arquitectura de la nube desplegada son un riesgo que el cliente es incapaz de gestionar, las políticas de uso podrían impedir realizar auditorías de seguridad en los servicios alojados en ella, etcétera. Asimismo están presentes otros riesgos clásicos en el acceso a servicios a través de la red como el robo o alteración de la información privada durante su tránsito por la red, la suplantación de identidad de los usuarios, o el acceso a recursos no autorizados.

Con el objetivo de que cualquier organización sea capaz de desplegar sus propios servicios de Computación en la Nube se creó OpenStack [9]. Este proyecto Open Source contiene el software completo para la puesta en marcha de un entorno de Computación en la Nube de tipo 'Infraestructura como Servicio' (IaaS) [4]. En ella se ofrecen como servicios la capacidad de cómputo de máquinas virtuales, su interconexión mediante el despliegue de elementos de red y la gestión de discos de almacenamiento. Por su carácter

abierto, se ha convertido rápidamente en un referente, por lo que empresas como HP, Cisco e Intel apuestan y colaboran en su mantenimiento. OpenStack no es un sistema monolítico, sino que está compuesto de módulos que gestionan determinados servicios. De estos, Keystone sería el central, pues se encarga de la autenticación y autorización de usuarios para el acceso al resto de servicios ofrecidos por los demás módulos.

Si comparamos las amenazas anteriores con la situación actual del proyecto OpenStack, vemos como estos riesgos han sido controlados. Para ello se implementan mecanismos como HTTPS [29] para la privacidad e integridad de las comunicaciones, se utilizan mecanismos seguros para la autenticación de usuarios - como transporte de credenciales sobre canales HTTPS u OAuth1.a [10] -, y existe un control de acceso a los recursos por medio de la asignación de roles de usuario y dominios de acceso. Aunque con esas tecnologías se da cobertura a los riesgos más clásicos, aún queda trabajo por realizar en el proyecto.

Un ejemplo concreto es la falta de un mecanismo de autenticación basado en Kerberos [11], que ha provocado que en organizaciones como el CERN [12] estén considerando implementar sus propias soluciones para integrarlo con OpenStack [13]. El motivo es que Kerberos es un mecanismo seguro de autenticación de usuarios, basado en el uso de claves simétricas, que ya es ampliamente utilizado por otros servicios de aplicación como SSH, FTP y SMTP. Adicionalmente, su uso permite el acceso a múltiples servicios con un único proceso de autenticación, lo que se conoce como *Single Sign On* (SSO). Además, mediante Kerberos Cross-Realm [17] se permite la autenticación de usuarios pertenecientes a organizaciones externas, siendo este proceso un caso concreto de implementación para lo que se conoce como 'acceso federado a servicios' o 'federación de identidades'.

En la federación de servicios, un usuario se encuentra afiliado a un proveedor de identidad (IdP en adelante), como puede ser, su universidad o un centro de investigación. El IdP se encarga de almacenar y gestionar la información personal del usuario. Este usuario tiene acceso a servicios ofrecidos por su organización (e.g. AulaVirtual, un servidor FTP, etcétera), pero también puede acceder a servicios ofrecidos por organizaciones ajenas siempre que tengan un acuerdo con su IdP. Esto es lo que ocurre, por ejemplo, en el proyecto eduroam [18] - el cual utiliza una tecnología diferente a Kerberos Cross-Realm, pero en ambos casos existen acuerdos entre IdPs y proveedores de servicios -. Este acceso es posible porque el servicio es capaz de delegar la autenticación del usuario al IdP mediante el uso de tecnologías de federación de identidad, suponiendo un gran beneficio para usuarios, que ya no necesitan ceder su información a otros servicios; para los administradores de sistemas; que ven su trabajo simplificado al no tener que gestionar altas de usuarios externos en sus sistemas, y para las propias organizaciones, que pueden

obtener ganancias mediante la monetización de dichos acuerdos de federación.

A día de hoy, la mayoría de tecnologías que permiten la federación de servicios están basadas en tecnologías web, como Shibboleth [30] y SAML [20]; o limitan su nivel de alcance al acceso a redes, como es el caso de eduroam y el protocolo EAP [25]. No obstante, existe una propuesta presentada por el IETF [31] para proporcionar acceso federado a servicios no orientados a web. La arquitectura propuesta tiene como nombre *Application Bridging for Federated Access Beyond Web* [26] (ABFAB) y surge como fruto del trabajo del proyecto Moonshot [32], por lo que se le suele denominar ABFAB/Moonshot.

Recapitulando lo anterior, la integración entre OpenStack y Kerberos supondría las siguientes ventajas:

- Que se dé respuesta a las solicitudes emitidas por parte de la comunidad para que exista un mecanismo de autenticación en OpenStack basado en la arquitectura Kerberos.
- OpenStack quedaría integrado dentro del abanico de servicios que permiten autenticación Single Sign On mediante el entorno Kerberos, con los beneficios de comodidad y seguridad que implica para los usuarios.
- Se abriría la puerta al uso de Kerberos Cross-Realm, lo que además de facilitar la experiencia de usuarios y administradores, podría implicar un beneficio económico en organizaciones que decidan ofrecer sus servicios como plataforma de Cloud mediante acuerdos de federación.

Por todo esto, se ha decidido que el principal objetivo de este proyecto sea la creación de un mecanismo para el módulo de gestión de identidades Keystone que permita la autenticación de usuarios mediante el uso de Kerberos. Tras su desarrollo y prueba se procederá a su publicación para su libre uso por parte de la comunidad. También se explorará la posibilidad del uso del mecanismo para la autenticación de usuarios federados mediante sistemas como Kerberos Cross-Realm mediante un estudio teórico, o la integración con el entorno ABFAB/Moonshot desarrollado para OpenStack por la Universidad de Kent mediante una breve prueba de concepto.

En los próximos capítulos de este documento se realizará un estudio completo del proceso de desarrollo de este Trabajo Fin de Grado (TFG). En el capítulo 2, se exponen los conceptos, tecnologías y protocolos utilizados en el trabajo. El capítulo 3, presenta un análisis de los puntos claves que deben cumplirse para la consumación exitosa del proyecto, así como los criterios que guiarán su desarrollo. El capítulo 4, contiene el análisis de los requisitos de integración entre los módulos existentes en el escenario y el del desarrollo de

un modelo para el mecanismos de integración entre OpenStack y Kerberos. En el capítulo 5, se detalla la arquitectura que ha sido necesaria desplegar para dar soporte a la solución, así como una descripción de la implementación de dicha solución. En el sexto y último capítulo, se incluye una recapitulación del proyecto, una valoración del resultado obtenido y las opiniones sobre el trabajo futuro que podría derivarse de la solución obtenida.

2. Estado del arte

Esta sección sirve como base introductoria a las nociones y tecnologías que serán el sustento del resto del trabajo. El objetivo es que el lector adquiera una idea general de las definiciones de cada concepto y la motivación de su aplicación, sin pretender extenderse con profundidad en los detalles.

2.1. Autenticación de usuarios basada en Kerberos y GSS-API

A continuación se presenta el protocolo de autenticación de usuarios Kerberos que será el usado en la resolución del proyecto, utilizando para ello la interfaz Generic Security Service Application Program Interface (GSS-API) [28].

2.1.1. Kerberos

Kerberos [33] es tanto una arquitectura como un protocolo para la autenticación de servicios en la red, diseñado para aplicaciones de paradigma cliente-servidor. Fue creado por el Instituto de Tecnología de Massachusetts (MIT) [34] dentro del proyecto Athenea [35] para dar solución a un problema común: garantizar que los usuarios puedan acceder a servicios ofrecidos por los servidores distribuidos en una red insegura como puede ser Internet. Los servidores deben ser capaces de restringir el acceso a todo usuario no legítimo y de autenticar las peticiones de acceso a servicios.

De un entorno donde no se puede tener la certeza de que un terminal o una dirección de red identifique a un usuario en particular se pueden esperar tres tipos de riesgos [36]:

1. Un atacante podría ganar acceso a un terminal y pretender ser un usuario legítimo.
2. Se podría modificar la red para que las peticiones enviadas desde una terminal maliciosa pareciesen provenir de la terminal legítima.
3. El intercambio de información entre el servicio y un terminal legítimo podría ser espiado por el atacante para obtener información con la que realizar un ataque de repetición, a fin de obtener acceso o interrumpir las operaciones.

Kerberos consigue que un usuario se pueda autenticar ante un servicio evitando todos estos riesgos. Para ello, junto a usuario y servidor se introduce un elemento adicional,

el Key Distribution Center (KDC). Su función es la de aportar el material criptográfico necesario para que puedan ser autenticados los usuarios y servicios de un domino concreto, que en jerga de la arquitectura se denomina *realm*. Se trata de un componente lógico, cuya funcionalidad se encuentra dividida en dos servicios. El primero es el Authentication Server (AS), que mantiene la información de todos los usuarios del sistema y una secreto compartido con ellos. El otro elemento es el Ticket Granting Server (TGS), encargado de almacenar la información de los servicios y el respectivo secreto compartido. En ambos servicios es necesario que los secretos compartidos existentes queden pre-establecidos mediante un canal seguro ajeno a Kerberos.

La forma en que usuarios y servicios se registran en el KDC es mediante el concepto de *principal*. Los *principals* son identificadores formados bajo la estructura `componente1/componente2/.../componenteN@REALM`. Por lo general, en Kerberos se utilizan dos componentes llamados *primary* e *instance*, quedando así el *principal* definido por la estructura `primary/instance@REALM`.

- *Primary* es la parte central, y en el caso de los usuarios se trata de su nombre de usuario, y para los servicios se suele utilizar la palabra *host* - aunque no es obligatorio -.
- *Instance* califica a *Primary*. En el caso de un usuario, esto significa que puede indicar bajo qué rol está actuando el usuario - alicia/admin frente alicia/usuaria -, mientras que para servicios se reserva para su FQDN - host@servicio.um -.
- *REALM* es el nombre del *realm* Kerberos al que pertenece. En general, se trata del nombre del dominio escrito en mayúsculas - e. g. el nombre de dominio servicio.um pertenecería al *realm* UM.ES-.

Con estos elementos en mente, el proceso de autenticación mediante Kerberos y acceso al servicio es el mostrado en la figura 1.

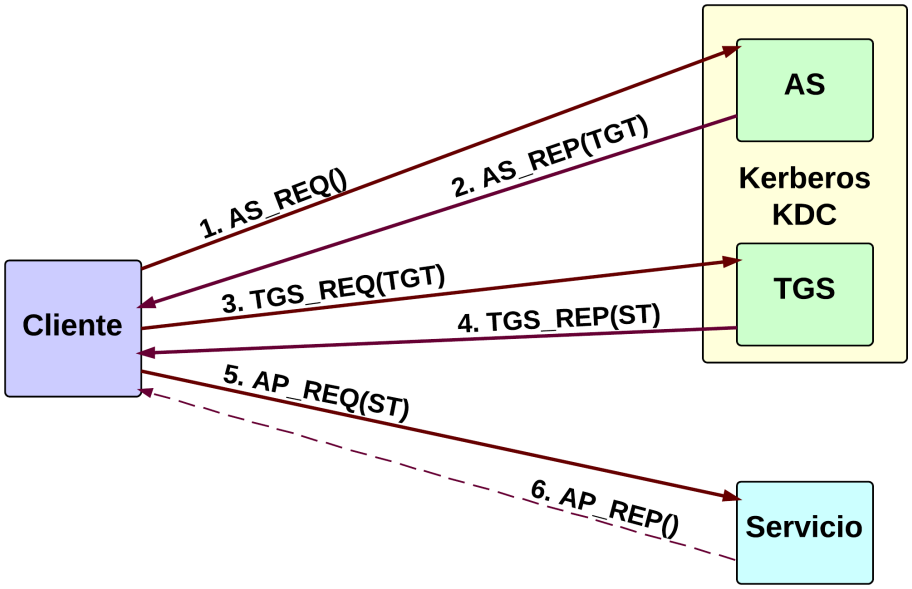


Figura 1: Autenticación con Kerberos y acceso al servicio.

- **Autenticación en el entorno Kerberos y obtención del Ticket Granting Ticket (TGT)**. Se realiza una única vez durante el tiempo establecido para la sesión, una vez haya expirado hay que repetirlo. En ella se intercambian dos mensajes:
  - 1. **AS_REQ**: del Cliente al AS. Su misión es solicitar en nombre de un usuario dado - *mediante su principal* -, la obtención de un TGT que sirva para una solicitud posterior al TGS de un ticket específico para el servicio. Dentro del mismo dominio o *realm*.
  - 2. **AS_REP(TGT)**: del AS al Cliente. Contiene un TGT para el usuario, cifrado con la clave compartida por ambos, por lo que solo si el usuario posee el secreto compartido, podrá conseguir el TGT para continuar el proceso..
- **Solicitud de un Service Ticket (ST)**. Es realizada una única vez por cada servicio accedido, durante el tiempo de sesión anteriormente comentado. Se compone de los mensajes:
  - 3. **TGS_REQ(TGT)**: del Cliente al TGS. El cliente presenta el TGT, extraído del mensaje anterior, como prueba de su identidad e indica a qué servicio desea acceder -utilizando el *principal* del servicio para ello-.
  - 4. **TGT_REP(ST)**: del TGS al Cliente. Se recibe como respuesta un ST cifrado con el secreto compartido entre TGS y el servicio.
- **Acceso al servicio**. Ejecutada una vez por sesión del usuario en el mismo servicio. Compuesta por:

- 3. **AP_REQ(ST)**: del Cliente al servicio. Finalmente, se envía el ST cifrado que se ha recibido en el mensaje anterior. El servicio comprobará la integridad y la validez del ST mediante la clave compartida que tiene con el TGS, y si son correctas se otorgará acceso al servicio.
- 4. **AP_REP()**: del servicio al Cliente, enviada de forma opcional. Para que el usuario pueda comprobar que efectivamente se trata del servidor legítimo y no uno malicioso, el servicio responde con una prueba de su identidad extraída del ST.

2.1.2. Generic Security Service Application Program Interface (GSS-API)

El hecho de ligar una aplicación con unos mecanismos de autenticación concretos implica que, en caso de querer ampliar la suite de métodos con nuevos mecanismos que puedan aparecer en el futuro, sea necesario la modificación de dicha aplicación junto a la implementación del nuevo método.

Para evitar esto se propuso una API denominada Generic Security Service Application Program Interface [28] (en adelante, GSS-API). Esta API permite implementar un sistema de autenticación con sencillez, entre clientes y servicios, que además resulta independiente del mecanismo de autenticación subyacente. Dicho mecanismo puede ser Kerberos [37], u otros que se han ido desarrollando con el paso del tiempo como IKE [38] o EAP [39].

El modo en que GSS-API consigue esto se basa en el concepto de intercambio de tokens GSS-API. En su interior se encapsula el material necesario para el mecanismo escogido, de forma que cliente y servidor deben limitarse a transportar estos tokens de un extremo a otro sin preocuparse por su contenido. El canal por el que se transportan los tokens es ajeno a GSS-API, siendo por lo general el protocolo utilizado por las aplicaciones para comunicarse entre ellas - aunque también podría ser otro diferente-. Las GSS-API pueden tener dos roles según quién inicie la negociación con el envío del primer token GSS-API: a la que realiza el primer envío se le denomina *GSS-API Initiator* y a la otra parte, *GSS-API Acceptor*.

Además de la autenticación, GSS-API permite añadir integridad y/o confidencialidad entre el cliente y el servicio. Por ello, se divide en dos los tipos de operación que puede realizar:

1. **Establecimiento del contexto de seguridad**: se realiza de forma inicial mediante la autenticación entre los extremos. Esta autenticación puede ser unidirec-

cional o bidireccional. Se hace uso de las operaciones `GSS_Init_Sec_Context` y `GSS_Accept_Sec_Context`. La primera es invocada por la aplicación que inicia el proceso de autenticación, lo que fija el rol de *GSS-API Initiator* a su implementación local de la GSS-API. La otra aplicación invoca a la función `GSS_Accept_Sec_Context`, funcionando su GSS-API con el rol de *GSS-API Acceptor*.

Estas funciones reciben como parámetros un GSS-API token - salvo en la primera invocación a la función `GSS_Init_Sec_Context` -, el identificador del mecanismo utilizado, y el conjunto de *flags* que configuran las opciones de la negociación - p. ej. el de autenticación mutua-. En respuesta, estas funciones generan un estado que puede ser `CONTINUE_NEEDED` si se necesita más información del otro extremo antes de finalizar, o `COMPLETE` si no es necesaria. Junto al estado, se devuelve el token GSS-API que debe ser entregado al otro extremo, aunque también puede ocurrir que no se devuelva ningún token si el otro extremo no requiere más material.

2. **Operaciones que permiten añadir integridad y confidencialidad al envío mensajes**: se emplean una vez se ha establecido el contexto de seguridad GSS-API. Para dotar de confidencialidad a un mensaje se utiliza la función `GSS_Wrap` que recibe como parámetro el mensaje y devuelve como resultado un token GSS-API que se puede enviar al otro extremo de forma segura. Cuando se recibe este token, el otro extremo llama a la función `GSS_Unwrap` que generará como resultado el mensaje original. De la misma manera se puede proporcionar integridad a los mensajes enviados haciendo uso del par de funciones `GSS_GetMIC` y `GSS_VerifyMIC`. El material criptográfico necesario para estas funciones de seguridad es derivado del proceso de autenticación anterior.

En general, una negociación GSS-API se puede resumir en el flujo de mensajes mostrado por la figura 2:

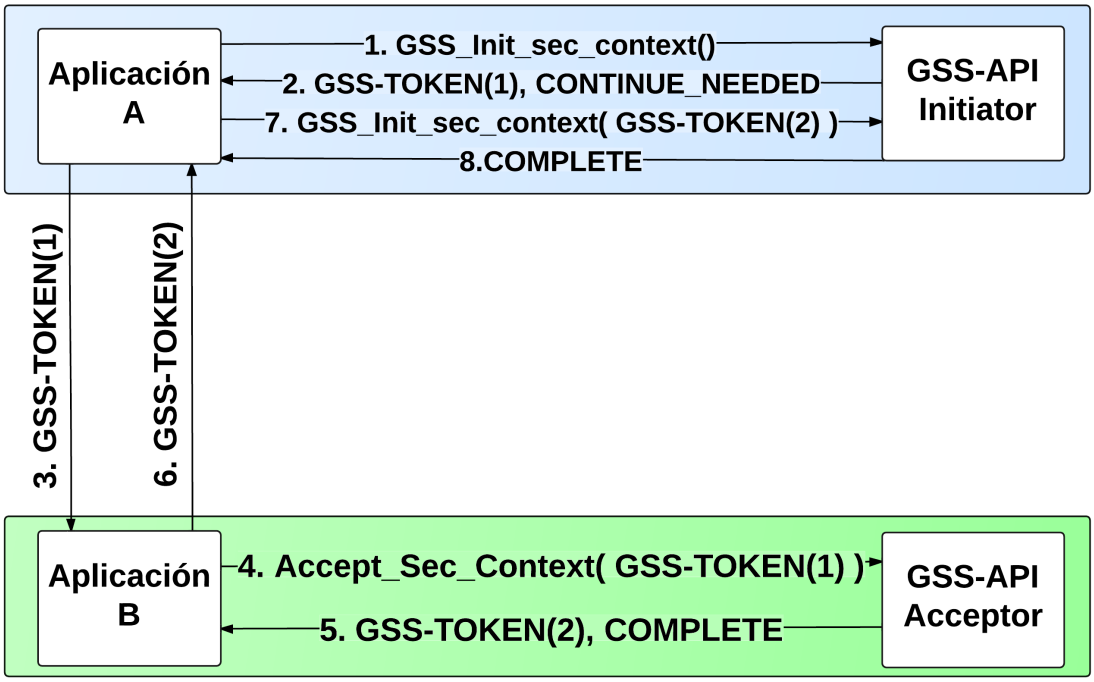


Figura 2: Autenticación de un usuario mediante la interfaz GSS-API.

1. La aplicación A invoca a la GSS-API mediante el método `GSS_Init_Sec_Context` y le pasa como argumentos el identificador del servicio, el identificador del mecanismos de autenticación que se desea utilizar, y ciertos flags como puede ser el de autenticación mutua requerida.

2. La GSS-API -en el papel de Initiator-, obtiene las credenciales necesarias del usuario para el mecanismo solicitado y genera un token GSS-API - `GSS-TOKEN(1)` -, que es entregado a la aplicación. Junto al token GSS-API se indica también un valor de estado: supongamos en este ejemplo que se muestra el estado `CONTINUE_NEEDED`, indicando así que espera recibir más material del otro extremo.

3. Se utiliza algún mecanismo de transporte ajeno a la GSS-API para transmistir el token entre las aplicaciones.

4. Cuando la aplicación B recibe el token GSS-API, se lo transmite a su implementación de la GSS_API mediante la llamada a la función `GSS_Accept_Sec_Context` que toma parámetros similares a `GSS_Init_Sec_Context`.

5. La GSS-API procesará el `GSS-TOKEN(1)`. Si consigue terminar de autenticar al usuario con éxito se devolverá otro token GSS-API - `GSS-TOKEN(2)` -, y mostrará un estado `COMPLETE` porque ha conseguido terminar de autenticar a la aplicación A.

6. Se envía el token `GSS-TOKEN(2)` a la aplicación A.

7. Esta vuelve a invocar al método `GSS_Init_Sec_Context` incluyendo esta vez el GSS-TOKEN(2) recibido.

8. Como esta vez la GSS-API ha podido autenticar al otro extremo, se devolverá el estado `COMPLETE`.

Tras esto, se garantiza que ambos extremos han sido autenticados. Desde este momento se pueden realizar invocaciones a la GSS-API para consultar por la identidad del otro extremo, o para obtener servicios de cifrado e integridad de los mensajes.

En este ejemplo se ha mostrado un caso breve del proceso de autenticación que coincide con el número de mensajes intercambiados al utilizar el mecanismo Kerberos. La GSS-API parte de la suposición de que el usuario se ha autenticado ante su KDC y tiene un ticket TGT. Tras la invocación a la función `GSS_Init_Sec_Context`, la GSS-API Initiator negociará con el KDC la obtención del ticket ST. Para ello se realizará el envío y la recepción de los mensajes Kerberos `TGS-REQ` y `TGS-REP`, seguido de un envío de los tokens GSS-API, que contienen los siguientes mensajes Kerberos:

- **`GSS-TOKEN(1)`**: mensaje Kerberos `AP_REQ` donde se incluye el ticket Kerberos ST.
- **`GSS-TOKEN(2)`**: mensaje Kerberos `AP_REP` que contiene una prueba de la identidad del servicio. El envío de este mensaje es opcional, sólo se realiza si durante la invocación a los métodos `GSS_Accept_Sec_Context` y `GSS_Accept_Sec_Context` se activa el *flag* de autenticación mutua.

Dependiendo de los mecanismos que se utilicen, el número de mensajes intercambiados varía. Para mecanismos que requieran más información puede ser necesaria varias iteraciones del proceso mostrado arriba. En general, el proceso de intercambio de tokens GSS-API se mantiene hasta que ambos extremos han mostrado el estado `COMPLETE` . La forma en que la API genera y procesa dichos tokens depende también del mecanismo usado para el establecimiento del contexto de seguridad.

2.2. Computación en la nube

La Computación en la Nube es un concepto cuya descripción resulta difusa y varía conforme las fuentes que se consultan dado su carácter novedoso. Una definición simplificada la describe como el software, plataforma o arquitectura que se vende como servicio a través de Internet [40]. De forma más técnica, el Instituto Nacional de Estándares y Tecnología [41] la define como un modelo ubicuo, cómodo y bajo demanda de acceso a la red, para compartir un conjunto de recursos de computación (redes, servidores, almacenamiento, aplicaciones y servicios) que pueden ser rápidamente desplegados o liberados con un esfuerzo mínimo de gestión o interacción por parte del proveedor de servicios [4].

Debe cumplir además con las cinco características básicas siguientes:

1. El autoservicio bajo demanda, el cual implica que un consumidor es capaz de abastecerse de recursos, como tiempo de cómputo de un servidor o almacenamiento en la red, bajo necesidad, de forma automática y sin tener que existir una iteración humana con el proveedor de servicios. Esto se puede hacer generalmente a través de un catálogo de servicios puesto a disposición del cliente.

2. Los recursos se encuentra disponibles a través del acceso por la red haciendo uso de mecanismos estándares y plataformas heterogéneas como teléfonos móviles, tabletas, portátiles, equipos de sobremesa, etcétera.

3. Se produce una agrupación de los recursos existentes en el centro de procesamiento de datos del proveedor para servir a múltiples consumidores, con diferentes recursos físicos y virtuales siendo dinámicamente asignados y reasignados de acuerdo a la demanda de los consumidor. Esto provoca una sensación de independencia de la localización en la que el consumidor no tiene control ni conocimiento de la situación exacta de sus recursos dentro de los servidores del proveedor. También gracias a esto, se aumenta la disponibilidad de los recursos al poder ser movidos virtualmente entre diferentes equipos en caso de fallo un sistema.

4. La elasticidad en una nube significa que los recursos pueden ser rápidamente aprovisionados y liberados, de forma automática en algunos casos, como respuesta a una variación considerable en su demanda de uso.

5. Se trata de un servicio medido. El uso de recursos puede ser monitorizado, controlado y reportado ofreciendo transparencia tanto para el consumidor como para el proveedor del servicio.

Los entornos de Computación en la Nube se pueden clasificar atendiendo a diferentes criterios. Si queremos referirnos a quién posee la propiedad de los recursos obtenemos cuatro categorías posibles:

1. La nube privada es aquella que pertenece a una única organización que hace uso de ella de forma privada con el objetivo de ayudar a alcanzar los intereses propios de ese colectivo y que utiliza una red de acceso restringida.

2. Existe una variante denominada nube en comunidad en la cual, la infraestructura es provista exclusivamente por una comunidad específica de organizaciones que tienen intereses en común. En este caso, el acceso se encuentra restringido a los miembros de las organizaciones pertenecientes a dicha comunidad.

3. Si el objetivo es ofrecer una infraestructura para el libre acceso del público general estamos ante una nube pública. Suelen pertenecer a organizaciones académicas, gubernamentales, de negocios o una combinación de estas.

4. Por último, existe un caso especial conocido como nube híbrida que se define por la composición de dos o más infraestructuras de nubes distintas que permanecen como entidades únicas pero que se unen bajo una tecnología que permite la portabilidad de datos y aplicaciones entre ambas.

Por otro lado, no todas los entornos de computación en nube ofrecen el mismo tipo de servicio. Existen, por tanto, tres niveles generales en función de lo ofrecido:

Software como servicio (SaaS): es el caso más común, siendo utilizado habitualmente por la mayoría de internautas. Ofrece a los usuarios acceso a aplicaciones software y sus bases de datos relacionadas. Se ofrecen sobre tecnologías webs. Algunos ejemplos son Google Docs [3], Gmail [6], Dropbox [5] y Spotify [2].

Plataforma como servicio (PaaS): otorga al usuario una plataforma de computación que suelen incluir un sistema operativo, un entorno de desarollo de un lenguaje de programación, bases de datos y servidores web. Está pensando para ser usado por desarrolladores que tenga la necesidad de poder escalar los recursos del entorno de trabajo para adaptarse a los cambios en las necesidades de la aplicación conforme evoluciona, pero que quieran abstraerse de la gestión de dichos recursos. Google App Engine[42] ofrece entornos para desarrollo de aplicaciones en Python[43], PHP[44], Java[45], Go[46] y una base de datos compatible con Mysql[47] llamada Cloud SQL[48]. Otro ejemplo podría ser Heroku[49], que permite el desarrollo de aplicaciones en Ruby[50], Java, NodeJS[51] y Python.

Infraestructura como servicio (IaaS): se trata de un nivel más bajo que el de PaaS. Aquí es el usuario el encargado de la gestión de la infraestructura del entorno desplegado. Se suele ofrecer, aunque no necesariamente a la vez, capacidad de almacenamiento básico (con gestión de los discos de almacenamiento), capacidades de cómputo (por medio de la creación de máquinas virtuales) y de despliegue de elementos de red (como routers, balanceadores de carga y cortafuegos). El servicio más conocido en este ámbito es el Amazon Web Service [52], que ofrece los servicios de almacenamiento y creación de máquinas virtuales. Por otra parte, Rackspace[53] ofrece también tanto despliegue de máquinas virtuales, como almacenamiento y gestión de la red de los equipos.

2.2.1. OpenStack

OpenStack [9] es un software de libre distribución que permite la creación de entornos de computación en nube tanto públicos como privados, con los objetivos de máxima escalabilidad y sencillez de despliegue y administración. Fundado originalmente por Rackspace Hosting y la NASA[54], ha sido acogido, desarrollado o financiado por una amplia gama de organizaciones, entre las que podemos citar algunas como AT&T, Canonical, HP, IBM, Cisco, Intel o VMware.

Está diseñado bajo una arquitectura modular en la que existe una serie de elementos lógicos independientes, encargados cada uno de un tipo de servicio concreto:

Nova: módulo de computación. Se encarga de la gestión de las pools de máquinas virtuales. Permite trabajar con distintos tipos de tecnologías de virtualización como KVM[55], XenServer[56] e Hyper-V[57].

Swift: módulo de almacenamiento. Es un sistema que permite el almacenamiento de objetos y ficheros en múltiples discos duros de forma que se garantiza la replicación y la integridad.

Cinder: módulo de almacenamiento de bloques. A diferencia del módulo Swift, está encargado del mantenimiento de dispositivos de almacenaje basados en bloques para usar junto a las instancias de máquinas virtuales creadas por el módulo Nova.

Neutron: módulo de red. Se encarga de la gestión de las redes y las direcciones IP.

Horizon: módulo de interfaz web. Permite el acceso al resto de servicios mediante una interfaz gráfica desarrollada sobre tecnologías web.

Keystone: módulo de identidad. Provee un directorio centralizado de usuarios y la lista de servicios a los que cada uno de ellos tiene acceso en concreto, así como sus roles definidos. Su tarea más habitual es la autenticación de usuarios y emisión de tokens que permitan el acceso al servicio.

Glance: módulo de imágenes. Se encarga de la gestión de imágenes de discos y servidores.

La dificultad técnica que implica el despliegue y la configuración de los módulos pueden suponer una importante barrera de acceso para aquellos administradores con poca experiencia en estos sistemas. Para un software que intenta acercar las tecnologías de la nube a cualquier empresa es necesario que esta dificultad sea minimizada. Por esta razón surgió el proyecto Devstack[58], en desarrollo desde 2011 por la OpenStack Foundation [59]. Aunque en palabras del propio equipo de desarrollo "no intenta ser un instalador general de OpenStack", permite desplegar de forma sencilla todos los nodos anteriormente mencionados para aquellos administradores inexpertos que quieran tener una toma de contacto.

La comunicación con los módulos mencionados anteriormente se realiza mediante el uso de REST [1]. Cada uno de ellos define una API de acceso a su servicio que pueden ser encontradas en [60]. La única excepción al uso de este modelo es Horizon, una interfaz web que permite el acceso a servicios como Nova, Swift y Keystone mediante el uso de un navegador web, lo que abstrae del uso de dichas APIs al usuario -Figura 3-.

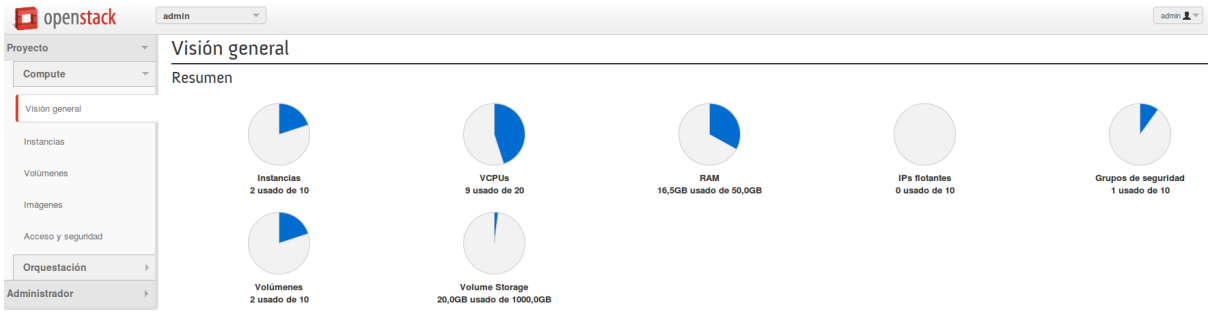


Figura 3: Interfaz Horizon mostrando estadísticas de uso

Para este proyecto resulta particularmente interesante el estudio del módulo Keystone, que, como hemos mencionado, es el encargado de ofrecer la gestión del servicio de identidad. Más concretamente, mantiene un directorio central de usuarios y sus relaciones con los servicios a los que pueden acceder. Este directorio puede estar implementado en forma de base de datos MySQL o de directorio LDAP[61].

[1]REST: Interfaz web simple que utiliza XML o JSON para codificar la información y los métodos HTTP get, post, put y delete para realizar peticiones.

Según el rol del usuario que lo utilice, Keystone ofrece distintas posibilidades. Como administrador, permite crear usuarios, proyectos y sus relaciones mediante los roles de cada usuario. Por otro lado, a los usuarios les otorga tokens Keystone con los que se pueden solicitar acceso otros servicios. Además, proporciona una lista de los servicios disponibles para cada usuario.

Existen dos tipos de tokens otorgados que pueden ser otorgados por el servicio Keystone. Los primeros serían los *unscoped tokens* con los que el usuario puede acceder a cualquier proyecto para el que esté autorizado. Sobre ellos se pueden solicitar que se limite la autorización del usuario para el acceso a un proyecto en particular, obteniéndose así un scoped token. Como su funcionalidad no afecta en el aspecto de autenticación ante el servicio, se usará sin pérdida de generalidad el término *token Keystone* para referirse a cualquiera de los dos.

A continuación se muestra un ejemplo de integración del módulo Keystone con el módulo de almacenamiento Swift. Se pretende simular el proceso de autenticación mediante la versión de la API v3 de Keystone y el posterior acceso al servicio Swift mediante la API v1 para obtener la lista de contenedores, siendo un contenedor el concepto equivalente a una carpeta en un sistema de ficheros común. Este proceso está ilustrado por la figura 4.

Figura 4: Autenticación con Keystone y acceso a Swift

En la comunicación se produce el siguiente intercambio de mensajes:

1. Un usuario solicita su autenticación antes de acceder a los servicios del sistema. Según la API de Keystone, para autenticar a un usuario es necesario hacer una petición HTTP POST al recurso `/v3/auth/tokens` del host donde se encuentre, a

a esta URL se la conoce como endpoint del servicio. En este ejemplo el endpoint es `keystone.OpenStack.um:5000`. El contenido que debe tener esta petición puede verse en la figura 5.

```
POST /v3/auth/tokens/ HTTP/1.1
Host: keystone.OpenStack.um:5000
Connection: keep-alive
Content-Length: 309
Content-Type: application/json
Accept: */*

{"auth": {
        "identity": {
            "methods": ["password"],
            "password": {
                "user": { "id": "2641
   e4fedcf64cdcab8a1533a16bd2e8",
                    "password": "alumno"}}}}}
```

Figura 5: Petición de autenticación a Keystone con API v3.

De este mensaje es interesante comentar varios aspectos. El atributo `methods` es un array que contiene el nombre de todos los mecanismos que debe pasar el usuario antes de poder ser autenticado; tras este atributo se deben incluir objetos con el mismo nombre que el definido en el interior de `methods` (vease `password`) y que contienen en su interior información específica para el mecanismo de autenticación especificado. Por ejemplo, en este caso sólo se ha seleccionado el mecanismo de `password`, y como información necesaria se incluye el identificador del usuario y su contraseña.

2. Keystone procesa esta petición, y responde con un mensaje HTTP 201 indicando que se ha autenticado con éxito al usuario. Incluye la cabecera **X-Subject-Token** que contiene el token de acceso a los servicios de OpenStack para los que el usuario tiene autorización.

```
HTTP/1.1 201 Created
X-Subject-Token: MIIe1AYJKoZIhvcNAQcCoIIexTCCHsECAQExCTA
HBgUrDgMCGjCCHSoGCSqGSIb3DQEHAaCCHRsEgh0XeyJ0b2tlbiI6IHs
ibWV0aG9kcyI6IFsia2VyYmVyb3NCYXNpYyJdLCAicm9sZXMiOiBbeyJ
pZCI6ICI5ZmUyZmY5ZWU0Mzg0YjE4OTRhOTA4NzhkM2U5MmJhY[....]
Vary: X-Auth-Token
Content-Type: application/json
Content-Length: 256

{"token": {"
  issued_at": "2013-09-04T22:26:20.863587Z",
  "extras": {},
  "methods": ["password"],
  "expires_at": "2013-09-05T22:26:20.863567Z",
  "user": {
    "domain": {
        "id": "default",
        "name": "Default"
    },
  "id": "2641e4fedcf64cdcab8a1533a16bd2e8",
  "name": "admin"}
  }
}
```

Figura 6: Respuesta a la autenticación con Keystone con API v3.

En la figura 6 podemos ver cómo se incluye el token en la cabecera `X-Subject-Token` y meta-información en el cuerpo del mensaje: periodo de validez, método por el que ha sido obtenido, usuario al que le pertenece, etcétera. El token es de una longitud considerable (en el ejemplo aparece abreviado por cuestion de espacio) ya que contiene mucha información sobre el usuario y además se encuentra firmado por el módulo Keystone. Gracias a esto, cuando es presentado ante otra entidad esta puede verificar su integridad sin necesidad de consultar con el Keystone, aumentando así la escalabilidad del servicio.

3. Con el token en su poder, el usuario puede solicitar acceso al resto de los módulos. En este caso accede al módulo Swift y le solicita que le muestre su lista de contenedores activos, de nuevo mediante una petición HTTP. Incluye además el token obtenido en el mensaje 2 como cabecera -Figura 7-.

```
GET /v1/AUTH_0abaeb222f024f8cbf3afcae43d823c6?format=json
   HTTP/1.1
Host: keystone.OpenStack.um:8080
Accept-Encoding: identity
X-Auth-Token: MIIe1AYJKoZIhvcNAQcCoIIexTCCHsECAQExCTAHBg
UrDgMCGjCCHSoGCSqGSIb3DQEHAaCCHRsEgh0XeyJ0b2tlbiI6IHsibW
V0aG9kcyI6IFsia2VyYmVyb3NCYXNpYyJdLCAicm9sZXMiOiBbeyJpZC
I6ICI5ZmUyZmY5ZWU0Mzg0YjE4OTRhOTA4NzhkM2U5MmJhYiIsICJuYW
[....]
```

Figura 7: Petición de servicio a Swift.

4. El servicio Swift valida el token recibido por el usuario y responde con un mensaje HTTP `200 OK` - Figura 8 -, que incluye además la información solicitada en el cuerpo del mensaje.

```
HTTP/1.1 200 OK
Content-Length: 48
Accept-Ranges: bytes
X-Timestamp: 1396454616.66239
X-Account-Bytes-Used: 0
X-Account-Container-Count: 1
Content-Type: application/json; charset=utf-8
X-Account-Object-Count: 0
X-Trans-Id: tx2ec9ca2c1be641d8ac4fb-0053563787
Date: Tue, 22 Apr 2014 09:33:59 GMT

[{"count": 0, "bytes": 0, "name": "contenedor"}]
```

Figura 8: Respuesta del servicio a Swift.

2.3. Federación de servicios

En los últimos años se ha visto como las Federaciones de Identidades han aumentado su nivel de popularidad por su capacidad de permitir a usuarios afiliados a una federación mediante un proveedor de identidad (IdP) el acceso a un servicio prestado por otra organización ajena denominada como proveedor de servicio (SP). Para lograr esto es

necesario que tanto IdP como SP pertenezcan a la misma federación. El proveedor de identidad es una organización que almacena información sobre usuarios que puede ser utilizada por aplicaciones de terceros con el consentimiento de sus dueños.

Podemos considerar que existen tres tipos de modelos en los que se pueden clasificar las federaciones de identidades [19]:

1. **Basada en Web**: hacen uso de tecnologías web como OAuth 2.0 [62], OpenID [22] o SAML [20]. Mediante estos sistemas los usuarios pueden utilizar una misma identidad para autenticarse ante diferentes plataformas web, evitando así tener que repetir los procesos de registro, centralizando la información personal en un punto y limitando el acceso que a ella tiene cada organización a la que se le permite el acceso. También se mejora así la calidad del servicio ofrecido ya que al evitar que el SP tenga que preocuparse por los mecanismos de autenticación del usuario le permite centrarse en ofrecer un mejor servicio.

2. **Basada en arquitecturas AAA**: se trata de un entorno usado para proveer de servicios de acceso a redes federadas. Las siglas AAA hacen referencia a los tres mecanismos básicos de control de la red:

   - Autenticación: consiste en el proceso de verificación de la identidad de un usuario mediante la presentación de unas credenciales de acceso válidas.
   - Autorización: es el proceso por el que se indican a qué servicios de la red tiene derecho el usuario una vez que la autenticación ha resultado positiva.
   - Contabilidad (Accounting): se encarga de recoger datos sobre el uso de los recursos empleados por el usuario durante su estancia en la red.

   En esta tecnología se basa la red eduroam [18], una federación que permite el acceso al servicio de red mediante una conexión WIFI a miembros de la comunidad universitaria siempre que se encuentren en alguna organización afiliada. En particular, eduroam hace uso del protocolo RADIUS [23, 24] que se encarga de la gestión de los tres mecanismos 'A' anteriores. Otro escenario de uso común de esta arquitectura se encuentra en las redes de telefonía móvil. El roaming, es decir, el uso por parte de un usuario de una red móvil externa a la del operador contratado se logra gracias al uso de la arquitectura AAA. En este caso se utiliza el protocolo Diameter [63], propuesto como evolución de RADIUS.

3. **Otras infraestructuras**: con la proliferación de los dos primeros modelos, ha aumentado el deseo de extender la federación a otros protocolos y servicios diferentes

a los web o de control de acceso a redes; pero desafortunadamente no existía hasta hace poco una tecnología de uso general para dar soporte al despliegue de arquitecturas en el último modelo. Para solucionar esta carencia el IETF ha presentado la arquitectura *Application Bridging for Federated Access Beyond Web* (ABFAB) [26]. Una motivación para su creación es que, por ejemplo, muchos proveedores de servicios basados en web permiten un acceso federado basado en el uso de mecanismos convencionales como SAML. Sin embargo, es común que los usuarios prefieran usar aplicaciones de escritorio que no hagan uso de los protocolos web. ABFAB podría ayudar en este contexto como una especificación que permitiría la autenticación federada para una variedad de protocolos no basados en tecnologías web.

2.3.1. Application Bridging for Federated Access Beyond Web (ABFAB)

La arquitectura de **Application Bridging for Federated Access Beyond Web** (ABFAB) surge a raíz del trabajo realizado por el proyecto Moonshot [32] desarrollado por Janet, la Red de Investigación Educativa del Reino Unido, en colaboración con otros integrantes. El objetivo de este proyecto es la creación de una tecnología unificadora que extienda los beneficios de la identificación federada a servicios no basados en web.

La arquitectura ABFAB se basa en el uso de mecanismos de seguridad y protocolos como EAP [25], GSS-API, SAML y el uso de arquitecturas AAA. Está compuesta por tres elementos principales: la aplicación cliente, que representa el deseo del usuario final de acceso a un servicio; el *relaying party* (RP) que controla el servicio que el usuario quiere utilizar; y el proveedor de identidad que es capaz de verificar las credenciales del usuario y además es el responsable de la distribución de información de autorización (mediante la definición de roles o atributos del usuario) hacia el RP.

Para aclarar mejor todos estos conceptos se muestra un escenario de uso como ejemplo. Supongamos que Alicia (el cliente) es una profesora perteneciente a la Universidad de Murcia (IdP) que quiere acceder al servicio de computación (RP) de la Universidad de Kent (SP) mientras se encuentra allí de visita. Ambas universidades pertenecen a la misma federación de identidad AAA (eduroam), y el servicio de computación hace uso del protocolo SSH como mecanismo de acceso. El servicio necesita comprobar que Alicia es quien dice ser (autenticación) y que tiene el rol necesario (autorización) para acceder al servicio, ya que solo los profesores y no los alumnos pueden hacer uso de él.

El flujo de mensajes intercambiados para el acceso al servicio en este escenario se encuentra representado en la figura 9. A continuación, se detalla cuál es la función de

cada uno de estos mensajes.

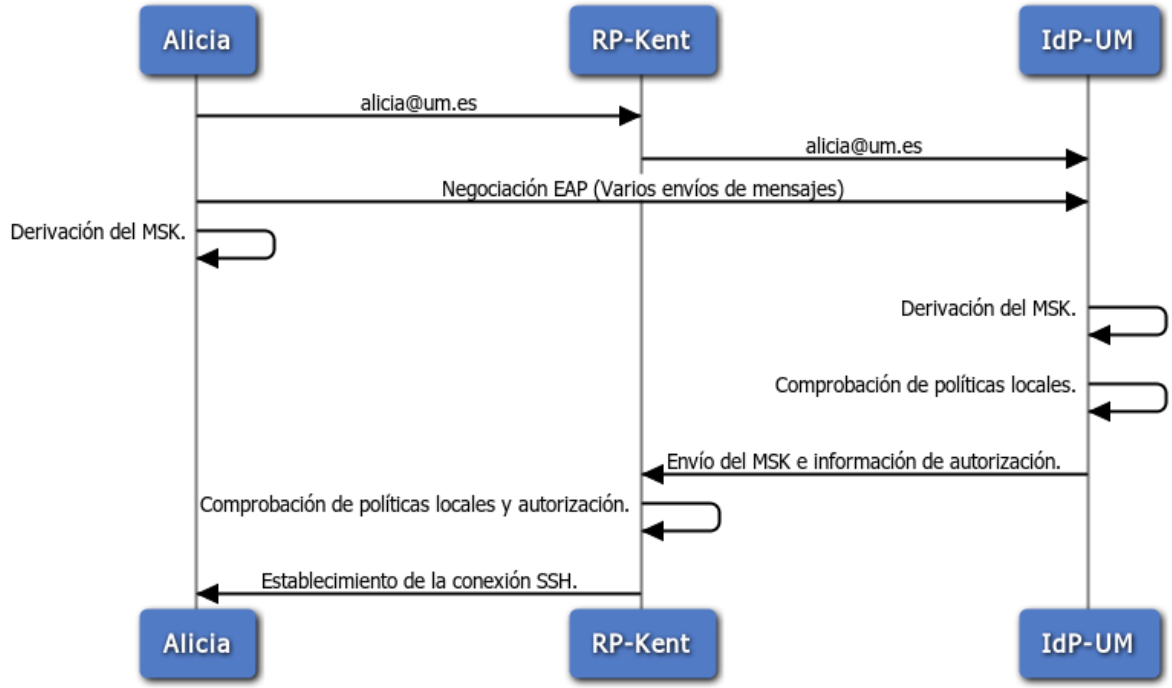


Figura 9: Ejemplo de acceso a un servicio mediante ABFAB.

- El cliente de Alicia comienza la conexión SSH con el servicio indicando que se desea realizar una autenticación basada en GSS-API con el mecanismo GSS-EAP [39]. Tras esto se produce el envío del primer paquete EAP contiene su identificador (alicia@um.es) hacia el RP.

- Como el RP no conoce nada sobre Alicia, obtiene la información sobre su IdP del dominio del correo entregado, *um.es*, y retransmite el paquete hacia el IdP de la Universidad de Murcia. Esto es posible gracias a que existe una red federada entre ambas instituciones. Sobre esta red federada se ejecuta el protocolo RADIUS que es el encargado de obtener la información sobre Alicia en el IdP de la Universidad de Murcia y traerla hasta el RP de Kent.

- Cuando el IdP recibe el paquete EAP comienza el proceso de autenticación del usuario, el cual puede consistir en múltiples rondas de envíos de mensajes. Todos estos mensajes son transportados usando al RP de intermediario.

- Cuando la autenticación se completa con éxito, tanto Alicia como el IdP de la Universidad de Murcia son capaces de derivar un secreto compartido *MSK*. Antes de que el IdP informe al RP de que la autenticación ha tenido éxito el IdP comprueba

la lista de políticas de la Universidad de Murcia para saber si cliente y RP están autorizados para realizar la acción solicitada, y además obtiene la información sobre la identidad de Alicia que debe ser retransmitida hacia el RP.

- Tras esto, el IdP envía al RP la MSK y la colección de atributos seleccionados con la información sobre Alicia haciendo uso de SAML. Esta información se transporta sobre el protocolo AAA utilizado para conectar a la universidades. En particular, se hace uso del protocolo RADIUS, que transporta la información sobre un nuevo atributo definido para codificar sentencias SAML [64].

- Una vez recibido la información, el RP comprueba en sus políticas locales si Alicia está autorizada a utilizar el servicio utilizando la información proporcionada por su IdP.

- Si todo es correcto, se establece la conexión con Alicia.

2.3.2. Kerberos Cross-Realm

Una revisión de la versión 5 de Kerberos [65] introduce un mecanismo por el que los usuarios de un *realm* pueden acceder a servicios de otros *realms* no gestionados por su KDC. Esto se consigue mediante el establecimiento de relaciones de confianza entre los distintos KDCs utilizando para ello secretos compartidos. Las relaciones pueden ser unidirecionales si los usuarios de un *realm* pueden acceder a los servicios del otro pero no al revés, o bidireccionales si ambos usuarios pueden acceder a los servicios del otro *realm*.

Existen dos tipos de arquitectura según la forma en que se establecen las relaciones de confianza, la arquitectura jerárquica y la arquitectura adhoc. En la primera, cada KDC conoce únicamente el siguiente salto en el camino hasta el servicio en el *realm* ajeno al que se quiere obtener. De esta forma se simplifica el número de relaciones de confianza existentes. En la arquitectura adhoc, se establecen las relaciones de confianza con todos los KDCs. Esto hace que sea un modelo más complejo de mantener pero elimina los riesgos de seguridad que podrían aparecer al usar KDCs desconocidos como pasos intermedios [66].

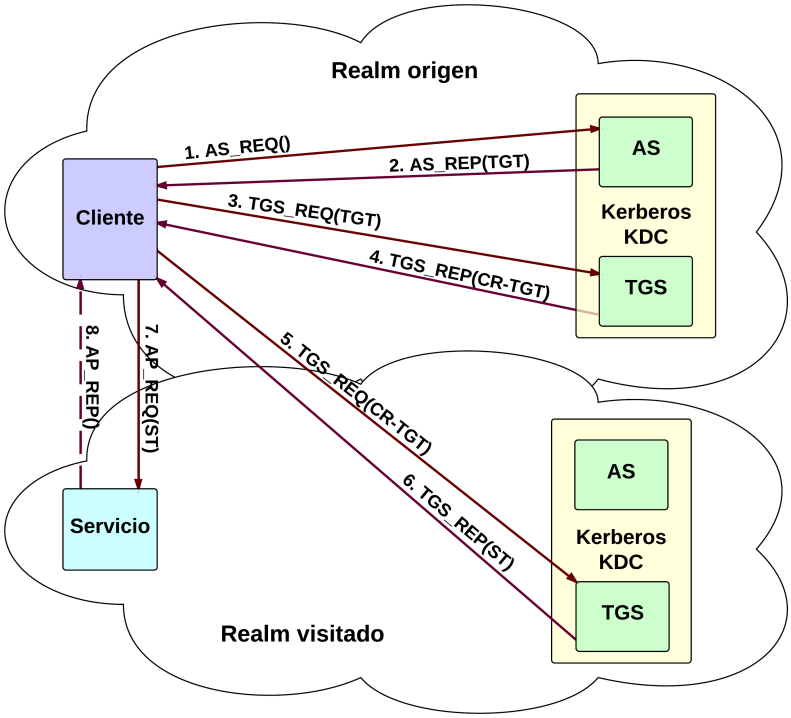


Figura 10: Acceso a un servicio mediante Kerberos Cross-Realm

En la figura 10 se muestra un ejemplo del proceso por el que un usuario consigue acceso a un servicio en un *realm* ajeno. La función de los mensajes empleados es la siguiente:

- **Mensajes 1 y 2**: el cliente se autentica en el entorno -de la forma habitual explicada en el apartado 2.1.1-, y obtiene su TGT.
- **Mensajes 3 y 4**: en lugar de solicitar un ST para el servicio, se solicita un TGT al TGS del *realm* origen para el TGS del *realm* visitado. Este TGT especial se muestra en la imagen anterior bajo el nombre CR-TGT.
- **Si la arquitectura no es ad-hoc:** por cada KDC en la cadena de confianza, el usuario presenta el CR-TGT del KDC anterior y obtiene el CR-TGT del siguiente KDC hasta llegar al KDC del servicio.
- **Mensajes 5 y 6**: con el CR-TGT, el cliente solicita al TGS del *realm* visitado un ST para el servicio que desea utilizar.
- **Mensajes 7 y 8**: el usuario accede al servicio utilizando el ST, y opcionalmente, el servicio envía una prueba de autenticación.

Cabe destacar, que en este ejemplo se ha utilizado una arquitectura con dos únicos *realms* en la cadena de confianza. En otros escenarios, la longitud de dicha cadena puede

ser arbitraria. En ese caso, el intercambio de mensajes se ampliaría incluyendo un envío del ticket CR-TGT conseguido en el *realm* anterior, hacia el siguiente elemento de la cadena de confianza. Este proceso se repite por cada relación de *realms* intermedios hasta llegar al destinatario del servicio.

3. Análisis de Objetivos y Metodología

3.1. Análisis de objetivos

Como hemos visto en la introducción, cada vez más organizaciones optan por usar los sistemas de computación en la nube ofrecidos por soluciones como Openstack. Keystone, su módulo de gestión de identidades, implementa diferentes mecanismos de autenticación. La versión 2 de la API para el servicio de Identidad soporta la autenticación mediante el par nombre de usuario-contraseña y basada en tokens - similar a una clave de acceso maestra para administradores -. En febrero de 2013 se presentó la versión 3 [67] que permite la autenticación mediante el protocolo OAuth [68].

Estos mecanismos cumplen con el objetivo de autenticar a los usuarios en la mayoría de escenarios. Sin embargo, algunas organizaciones, como el CERN, han manifestado su interés en poder hacer uso de Kerberos como mecanismo de autenticación [13] y aprovechar así la infraestructura Kerberos que ya tienen desplegada. Aunque Openstack se ha hecho eco de estas peticiones y está planificada su futura integración, aún no hay una fecha concreta para ello [69].

En base a esta situación, se ha decidido que la motivación de este proyecto sea incorporar un proceso de autenticación basado en Kerberos con el software Openstack. Con esta integración se podrán aprovechar la infraestructuras para Kerberos ya desplegadas en muchas organizaciones. También permitirá dotar a Openstack de un soporte de Single Sign On más global, pudiendo ser utilizadas otras aplicaciones como SSH o FTP sin necesidad de repetir el proceso de inicio de sesión. Además, se habilitará que usuarios federados tengan acceso a Openstack mediante el uso de Kerberos Cross-Realm.

De la mencionada motivación se derivan los siguientes objetivos del proyecto:

- Análisis del funcionamiento del entorno OpenStack, y en particular, del módulo Keystone. Estudio de la arquitectura Kerberos y uso de la GSS-API.
- Diseño de un mecanismo de autenticación mediante Kerberos para Openstack, realización de su implementación y prueba de funcionalidad mediante la adaptación para su uso del cliente del servicio Swift.
- Estudio teórico de las capacidades de SSO y federación mediante Cross-Realm y prueba de concepto para la integración Kerberos en el entorno ABFAB/Moonshot de OpenStack.

3.2. Metodología

Existe una falta de documentación clara, concisa, sin errores y actualizada sobre el funcionamiento del software Openstack, sus interfaces y las opciones de extensibilidad disponibles. Todo este conjunto de problemas, sumado a la inmersión desde cero en un software de elevada complejidad, ha provocado la aparición de múltiples complicaciones y contratiempos en el planteamiento del proyecto. Ante esta situación se ha adoptado la siguiente metodología:

1. Lectura y compresión de la información sobre el entorno Openstack.
2. Análisis de los mecanismos de autenticación habitualmente utilizados (Kerberos, GSS-API, canales HTTPs, OAuth).
3. Análisis del código fuente de los servicios y clientes de Openstack.
4. Diseño de un desarrollo básico que incluya las modificaciones necesarias en Openstack para su integración con Kerberos.
5. Una vez finalizado el desarrollo, despliegue de un escenario de pruebas que demuestre su viabilidad.
6. Documentación del trabajo realizado.
7. Cierre y publicación del código fuente implementado.

4. Diseño de la solución

4.1. Requisitos de la integración entre OpenStack y Kerberos

Para que una aplicación pueda realizar el proceso de autenticación de un usuario mediante Kerberos es necesario que esta sea '*Kerberizada*', es decir, que tenga la capacidad de generar y procesar los mensajes y tickets de Kerberos. La implementación de todos esos métodos supone crear una dependencia con el mecanismo. Sin embargo, esto no constituye un problema real, ya que se puede eliminar haciendo uso de la GSS-API.

Con ella se puede alternar entre distintos métodos de autenticación - Kerberos entre ellos -, sin tener que cambiar la estructura interna de la aplicación gracias al uso de una interfaz genérica. Esto supone que en caso de una alteración del protocolo no haya que readaptar el mecanismo, y que la extensión a otros protocolos de autenticación sea sencilla mediante la reutilización de la arquitectura de este.

Hay dos elementos necesarios para que la GSS-API con el mecanismo Kerberos pueda llevar a cabo el proceso de autenticación:

- Que en los equipos donde se haga uso de ella existan ficheros de configuración de Kerberos, con los datos del Centro de Distribución de Claves (KDC) del *realm* que gestione la identidad de los usuarios. En esta información se incluye tanto direcciones de red, como secretos compartidos utilizados en el proceso de autenticación.
- Que se establezca un canal sobre el que transportar los tokens generados por la GSS-API en los que se encapsulan información relacionada con el protocolo de autenticación empleado, siendo estos, en nuestro caso, tickets Kerberos.

El primer punto se soluciona en el equipo cliente de la aplicación al instalar y configurar el cliente Kerberos. Este cliente es necesario para iniciar sesión en el entorno Kerberos. Por otro lado, para el equipo servidor es suficiente con disponer de los ficheros de configuración de Kerberos, sin necesidad de instalar también el cliente. La información necesaria sobre el realm Kerberos para la configuración puede ser obtenida directamente del administrador del sistema, o por DNS [70] [71].

Para el intercambio de los tokens entre los dos extremos de la GSS-API se utiliza el mecanismo REST HTTP que implementa el módulo Keystone. Además de esto, por supuesto, se requiere el despliegue de un KDC que contenga los usuarios que han de autenticarse y una entrada para el módulo Keystone como servicio.

4.2. Integración de Openstack y Kerberos

Teniendo claro los requisitos previos, vamos a detallar a continuación cómo se realizaría el proceso de acceso a un servicio de Openstack con nuestro mecanismos. En concreto usaremos, sin pérdida de generalidad, al servicio Swift como referencia. El escenario completo consistirá en la autenticación en el entorno Kerberos para la obtención del TGT, seguido de la obtención de un token por parte del módulo Keystone - utilizando para ello el material Kerberos con ayuda de la GSS-API -, y por último, el acceso al servicio.

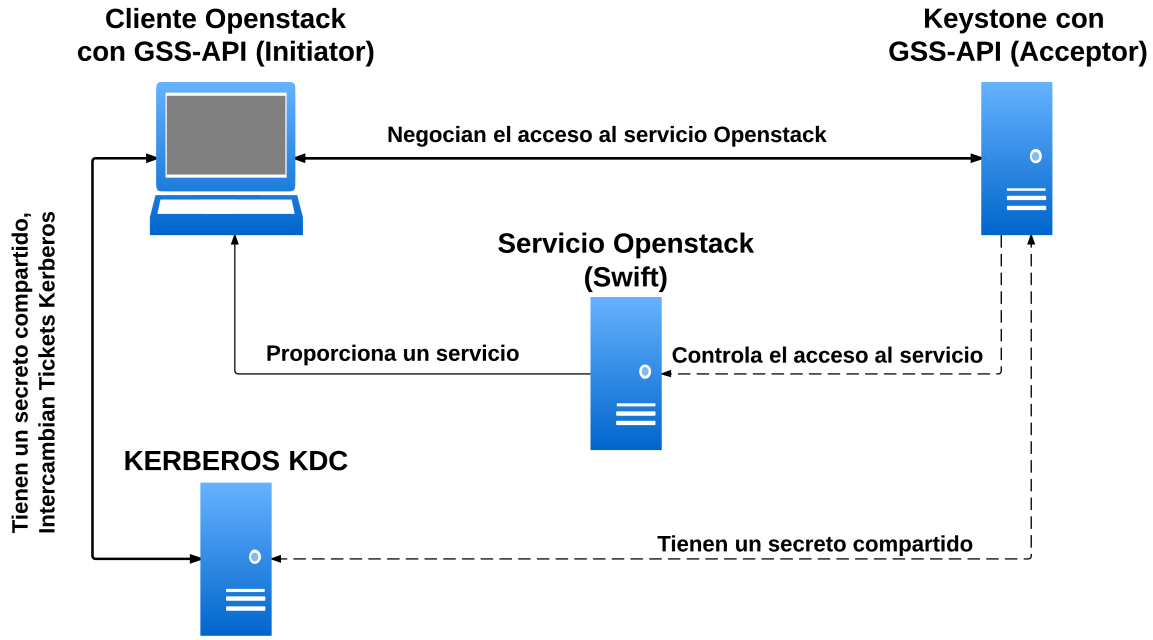


Figura 11: Relaciones entre los elementos de la arquitectura.

En la figura 11 podemos ver las relaciones que existen entre los elementos de la arquitectura. El cliente forma parte de un *realm* Kerberos, lo que implica que tenga un secreto compartido con el KDC. Además, ese cliente desea acceder al servicio OpenStack (Swift) cuyo acceso es controlado por el módulo Keystone mediante la emisión de tokens. El módulo Keystone también pertenece al *realm* del usuario e igualmente tiene una clave común con el KDC.

Las solicitudes al módulo Keystone tienen una estructura común independientemente del mecanismo de autenticación que se emplea. Dicha solicitud es el resultado de encapsular en un mensaje POST HTTP un objecto JSON con la información del proceso.

```
{
  "auth":{
    "identity":{
      "methods":["Method1","Method2", ... , "MethodN"],
      "Method1" : { Información sobre el método 1},
      "Method2" : { Información sobre el método 2},
                  ...
      "MethodN" : { Información sobre el método N}
    }
  }
}
```

Figura 12: Contenido de una solicitud de autenticación genérica

- El array `methods` contiene el nombre de todos los métodos de autenticación por los que el usuario debe pasar para ser autenticado [72].
- Por cada método presente en `methods` existe un objeto con su mismo nombre ('Methods1', 'Methods2', ..., 'MethodsN'), que contiene la información necesaria para autenticar al usuario.

En nuestro escenario hemos decidido denominar al método como 'KerberosBasic', por lo que el mensaje tendría un aspecto como el siguiente:

```
{
  "auth":{
    "identity":{
      "methods":["KerberosBasic"],
      "KerberosBasic" : { Información para la autenticación}
    }
  }
}
```

Figura 13: Contenido de una solicitud de autenticación mediante Kerberos

Por otra parte, Keystone puede responder de dos maneras ante una solicitud de autenticación:

- **Con una respuesta final**, indicando si la autenticación ha tenido éxito o no. En el caso de ser afirmativa, se incluirá un token Keystone de acceso a los servicios de OpenStack para el usuario autenticado.
- **Con una repuesta provisional**, indicando que es necesario proporcionar más información para continuar con la autenticación. En este caso, junto a la respuesta se puede incluir información que el usuario puede utilizar, bien para continuar con su autenticación, o bien para autenticarse ante el otro extremo.

Tras conocer esto, se ha diseñado el mecanismo de autenticación como un proceso divido en las tres partes siguientes:

1. **Descubrimiento**: el cliente comprobará que el mecanismo de autenticación KerberosBasic está disponible. Keystone asignará al cliente un identificador de sesión para agrupar los mensajes HTTP del resto de la negociación y le dará su *principal* en el KDC.
2. **Negociación**: con la información recibida en el paso anterior y utilizando la GSS-API como abstracción, se realiza una autenticación mutua entre cliente y Keystone.
3. **Finalización**: si la autenticación ha sido exitosa el cliente recibe un token por parte del módulo Keystone. Este token debe ser presentado junto a la petición de acceso al servicio OpenStack para lograr la autorización de acceso.

En apartados siguientes se muestran los detalles de cada una de las etapas de la negociación.

4.2.1. Fase 1: Descubrimiento

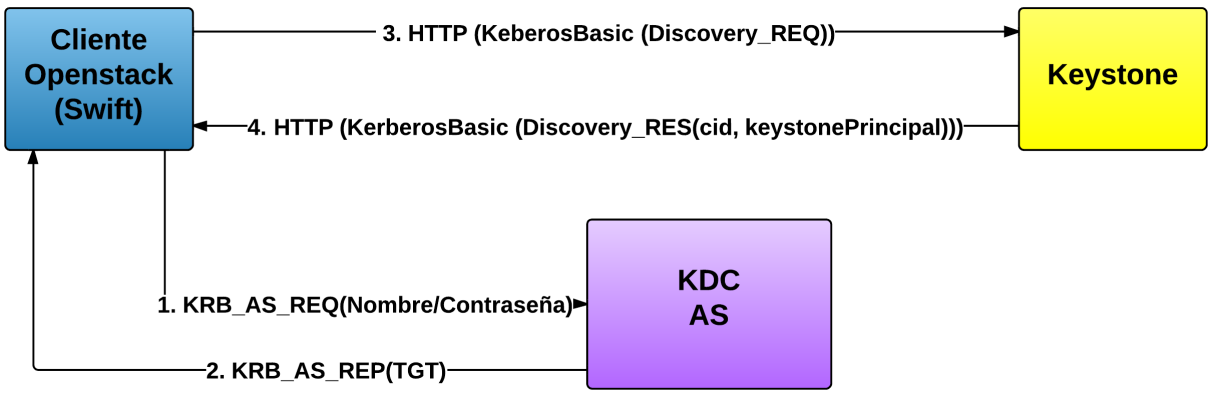


Figura 14: Fase de descubrimiento

En la primera fase existen dos objetivos principales:

1. Iniciar sesión en el entorno de Kerberos para la obtención del ticket Kerberos TGT. En la figura 14 está representado por los mensajes 1 y 2. Este proceso de inicio de sesión aparece simplificado para ayuda del lector, ya que realmente no se envían los credenciales al AS, tal y como se ha explicado en la sección de Kerberos en el Estado del Arte.

2. Comprobar que el mecanismo KerberosBasic se encuentra disponible en el Keystone al que estamos accediendo. En caso afirmativo, obtener un identificador de sesión - denominado cid - y el identificador del *principal* del Keystone en el KDC (Mensajes 3 y 4).

El primer objetivo se ha dejado fuera de la responsabilidad del cliente OpenStack. El motivo es porque se trata de un proceso que usualmente se encuentra automatizado con el inicio de sesión en el sistema. En el caso de que el usuario no tenga habilitada esta opción, bastaría con que ejecutara el comando `kinit` [2] junto a su nombre de usuario en el KDC. Con esto, el usuario queda autenticado dentro del entorno Kerberos y consigue el ticket kerberos TGT con el que puede solicitar acceso al resto de servicios registrados en el KDC sin necesidad de autenticarse de nuevo.

Para la segunda parte se emplea un mensaje KerberosBasic de tipo Discovery (3) que tiene la siguiente estructura:

```
{
"auth":{
  "identity":{
    "methods":["KerberosBasic"],
    "KerberosBasic" : {
      "phase": "discovery"
    }
  }
}
}
```

Figura 15: Fase de descubrimiento: Solicitud (3)

[2] `kinit`: Comando de inicio de sesión y obtención del ticket TGT en entornos Kerberos. Proporcionado mediante el software del cliente Kerberos junto a otros comandos como `klist` para ver la caché local de tickets Kerberos.

Como respuesta (4), contestará con un mensaje con la forma del mostrado en la figura 16. En él se incluye un identificador de sesión incluido en el campo `cid` que, por otra parte, es necesario, ya que HTTP es un protocolo sin estado. De esta manera, para mantener agrupados todos los mensajes pertenecientes a una sesión de autenticación, se requiere incluir algún tipo de información que relacione las peticiones entre sí. En servicios web se suelen utilizar cookies para este fin, pero en este caso se ha incluido un identificador único universal UUID [73] para simplificar la lógica de la aplicación. Este tipo de identificador tiene la propiedad de garantizar que su valor sea único de forma universal, evitando así la necesidad de gestionar las posibles colisiones entre identificadores otorgados a los usuarios. También se incluye en el campo `service` el principal del Keystone en el KDC que será empleado por el cliente para indicar ante qué servicio quiere autenticarse.

```
{
  "error": {
    "message": "Additional authentications steps required.",
    "code": 401,
    "identity": {
      "methods": ["kerberosBasic"],
      "kerberosBasic": {
        "negotiation": "<<Vacio>>",
        "cid": <<Identificador UUID de la sesión>>,
        "service": <<Principal de Keystone en el KDC>>  }
      },
      "title": "Unauthorized"
  }
}
```

Figura 16: Fase de descubrimiento: Respuesta para mecanismo correcto (4)

Con la recepción de este mensaje el cliente tiene la certeza de que el método Kerberos es soportado por la aplicación. En caso contrario se recibirá un mensaje similar al siguiente:

```
{
  "error" : {
    "message" : "Attempted to authenticate
          with an unsupported method."
    "code" : 401
    "identity": {
      "methods" : [<<Lista de métodos soportados>>]
    }
    "title" : "Unauthorized"
  }
}
```

Figura 17: Fase de descubrimiento: Respuesta para mecanismo inválido

Ante esta situación, el cliente es el encargado de implementar un sistema de recuperación y ofrecer al usuario la posibilidad de cambiar de mecanismo por otro de los presentes en la lista `methods` que sea capaz de manejar.

4.2.2. Fase 2: Negociación

El único objetivo de la fase 2 es completar el proceso de autenticación mediante Kerberos con ayuda de la GSS-API. Para ello, el cliente comienza creando un contexto de seguridad GSS-API (5). La GSS-API recibe como parámetros el identificador del mecanismo que se usa - Kerberos - y el *principal* del servicio ante el que se quiere autenticar - campo `service` en el mensaje 4 -.

Durante esta construcción del contexto, la GSS-API envía un mensaje `KRB_TGS_REQ` al módulo TGS del KDC (6), que contiene el ticket TGT - obtenido en el mensaje 2 -, y el *principal*. Como respuesta, la GSS-API obtiene un mensaje `KRB_TGS_REP` (7) que incluye un ticket Kerberos ST. Este ticket es envuelto en `GSS_TOKEN(KRB_AP_REQ(ST))` (8).

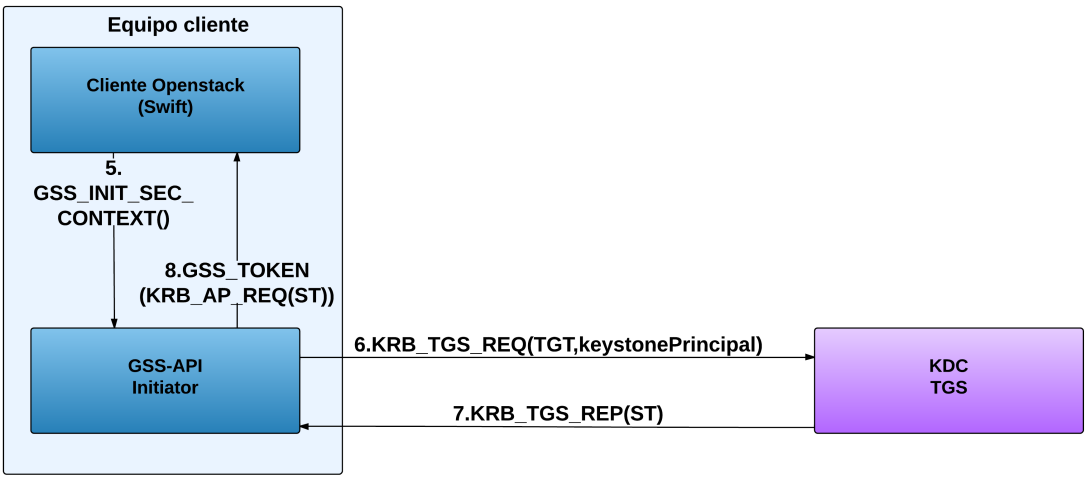


Figura 18: Fase de descubrimiento. Parte 1.

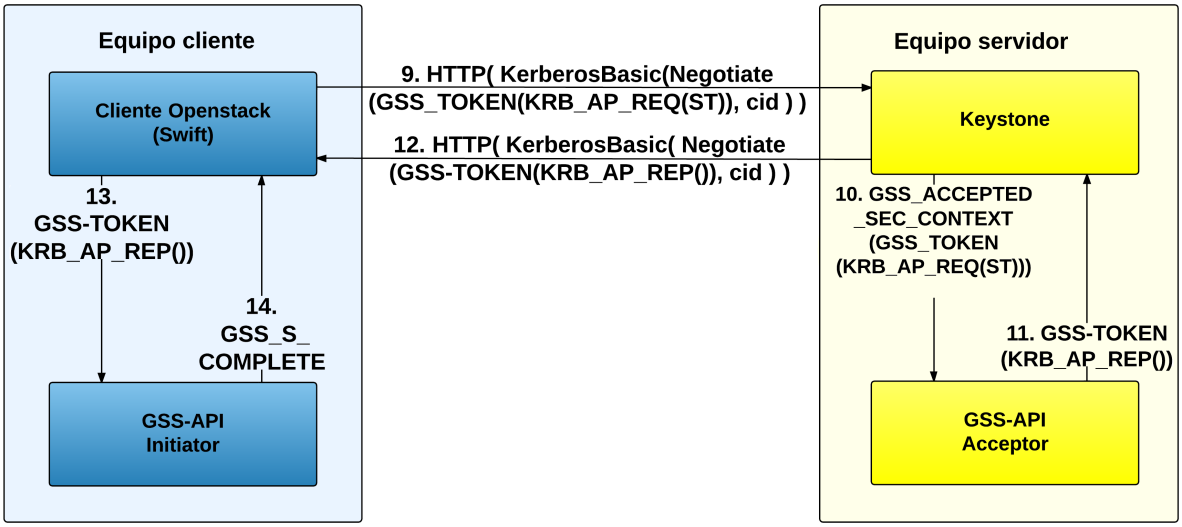


Figura 19: Fase de descubrimiento. Parte 2.

El token GSS-API es enviado al módulo Keystone mediante un mensaje KerberosBasic Negotiate (9) con una estructura como la siguiente:

```
{
  "auth": {
    "identity": {
      "methods": ["kerberosBasic"],
      "kerberosBasic": {
        "phase": "negotiate",
        "negotiation": <<GSS_TOKEN(KRB_AP_REQ(ST))>>,
        "cid": <<Identificador UUID de la sesión>>
      }
    }
  }
}
```

Figura 20: Fase de negociación: Solicitud (9)

Cuando Keystone reciba esta petición, comprobará mediante el uso de un repositorio de contextos GSS-API que aún no existe el contexto asociado al `cid` presentado y lo creará de manera similar a como lo hizo el cliente (10). En este caso, adicionalmente se le entrega el `GSS_TOKEN(KRB_AP_REQ(ST))` recibido desde el cliente. Durante este proceso la GSS-API verificará la validez del ticket Kerberos ST, quedando así autenticado el usuario ante el servicio.

Como también es necesario que Keystone se autentique ante el cliente, la GSS-API generará un mensaje `KRB_AP_REP` que el cliente puede verificar. Este mensajes se inserta en un token GSS-API, generándose finalmente el mensaje `GSS_TOKEN(KRB_AP_REP())` (11). A continuación, se transporta de vuelta mediante un mensaje KerberosBasic Negotiate:

```
{
  "auth": {
    "identity": {
      "methods": ["kerberosBasic"],
      "kerberosBasic": {
        "phase": "negotiate",
        "negotiation": <<GSS_TOKEN(KRB_AP_REP())>>,
        "cid": <<Identificador UUID de la sesión>>
      }
    }
  }
}
```

Figura 21: Fase de negociación: Respuesta (12)

Por último, el cliente recibirá el `GSS_TOKEN(KRB_AP_REP())` y se lo pasará a su GSS-API (13). Esta extraerá el mensaje `KRB_AP_REP()` que utilizará para autenticar al Keystone. Tras ello finalizará mostrando un estado `GSS_S_COMPLETE`, que indica que la autenticación ha sido exitosa (14).

Llegados a este punto, ambos extremos han sido mutuamente autenticados de forma satisfactoria.

4.2.3. Fase 3: Finalización

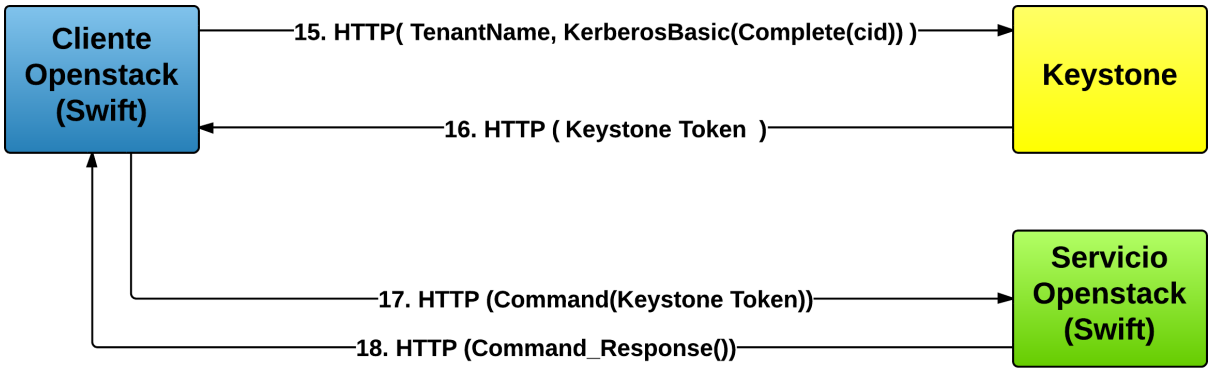


Figura 22: Fase de finalización

En esta fase ya sólo nos resta solicitar el token Keystone y acceder al servicio. Para ello, el cliente envía un mensaje que contiene opcionalmente el nombre del Tenant al que

quiere acceder y un mensaje KerberosBasic Complete como el de la figura 23, el cual únicamente contiene el `cid` de la negociación (15).

```
{
  "auth": {
    "tenantName": null,
    "identity": {
      "methods": ["kerberosBasic"],
      "kerberosBasic": {
        "phase": "complete",
        "negotiation": <<Vacío>>,
        "cid": <<Identificador UUID de la sesión>>
      }
    }
  }
}
```

Figura 23: Autenticación mediante kerberos completada: Solicitud del token Keystone

Keystone comprobará que el `cid` se ajusta a un contexto GSS-API válido, obtendrá de ahí el nombre del usuario autenticado y responderá con el Token (16). El cliente ahora puede acceder al servicio OpenStack ejecutando una llamada tradicional a la API en la que se incluye el token Keystone (17 y 18).

Llegados a este punto se da por finalizado el escenario de autenticación y acceso al servicio OpenStack.

4.3. Usuarios federados mediante Kerberos Cross-Realm

Para el caso en que el usuario sea federado, es decir, perteneciente a un *realm* de Kerberos diferente al del servicio OpenStack, no es necesario cambiar el diseño de la negociación. Esto es posible gracias a que el ticket Kerberos utilizado por Keystone para autenticar al usuario por medio de la GSS-API es el ticket ST, cuyo contenido no varía independientemente de si el usuario que lo obtiene pertenece o no al mismo *realm* que el servicio - Keystone en este caso -, y por ello, no hay que modificar su proceso de validación.

Llamaremos TGS Home al TGS del *realm* donde se encuentra registrado el usuario y TGS Visited al que tiene relación con el Keystone. La única diferencia en la secuencia

de pasos del mecanismo la encontramos en la fase de negociación. Antes de obtener el ST, la GSS-API deben encargarse de obtener el TGT para el acceso al TGS Visited -llamémoslo CR-TGT -, utilizando como autenticador para ello el TGT obtenido del TGS Home. La secuencias de mensajes entonces así:

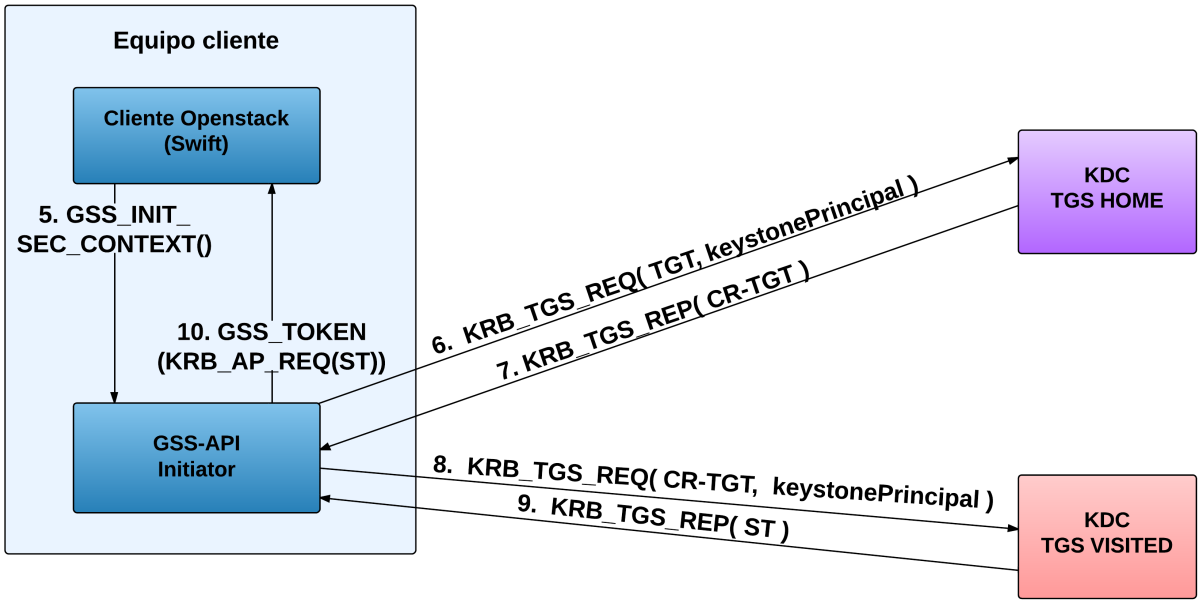


Figura 24: Ejemplo de fase de negociación con Cross-Realm con 2 dominios

Finalmente, el único problema que aparece con este escenario es el mapeo del usuario federado con un usuario del sistema OpenStack. Tras completar la segunda fase, el usuario ha sido autenticado con un identificador válido en su dominio de origen. Sin emabargo, este identificador no tiene por qué ser reconocido por el Keystone de la organización visitada, y por tanto, cuando se intente generar el token OpenStack para ese identificador se producirá un error, ya que no está dado de alta en los repositorios de usuarios de OpenStack de la organización visitada. Para solucionar esto hay dos posibles vías:

- Darlo de alta. Esta podría ser una solución válida si Keystone obtiene la lista de usuarios de su propia base de datos independiente. En otro caso - por ejemplo, si se obtienen del servicio LDAP de la organización- no sería viable ya que obligaría a registrar a un usuario completo, perdiéndose así las ventajas de la federación.

- Utilizar un usuario común previamente registrado - por ejemplo, usuario `federado` - para todos los usuarios pertenecientes al mismo *realm*. De esta forma se pierde granularidad, pero provoca menos problemas con la gestión de usuarios que en el caso anterior.

Es importante recordar que este análisis es solamente un planteamiento teórico de lo que se piensa que debería ocurrir. Aún sería necesario probarlo en un escenario práctico para constatar su validez. Esto se deja como trabajo futuro.

5. Implementación y despliegue

Esta sección está destinada a explicar cómo ha sido realizada la implementación del mecanismo KerberosBasic, y comentar las pruebas realizadas con él. Para ello, se describe el escenario desplegado junto a una guía de instalación del mismo, después se analizan las diferentes implementaciones realizadas y por último se muestran las pruebas de uso del escenario.

5.1. Despliegue y configuración del entorno de pruebas

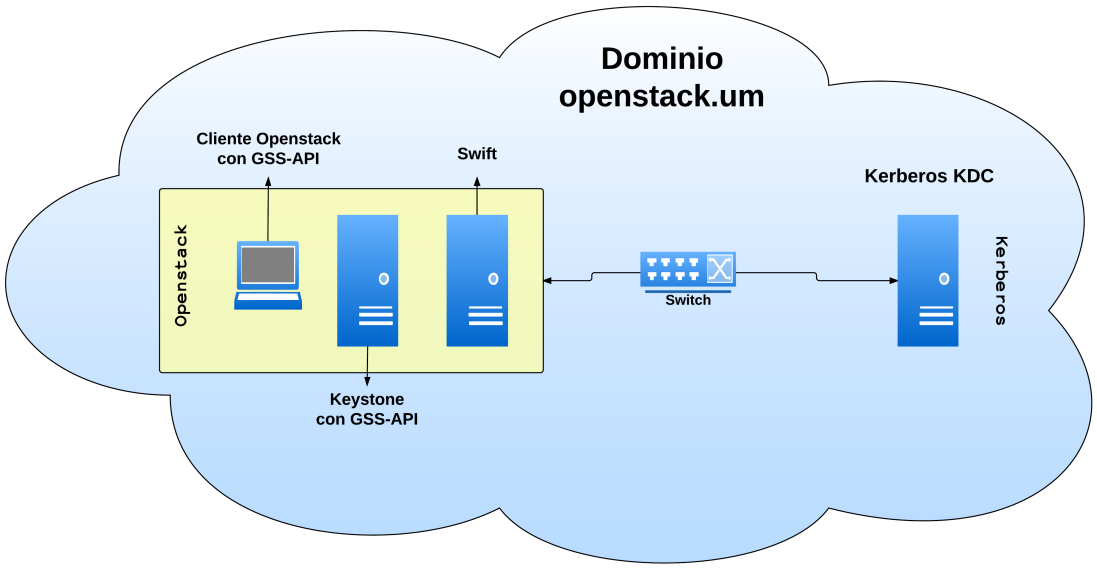


Figura 25: Escenario desplegado

La figura 25 muestra el escenario con el software y los equipos desplegados. Todos los servicios descritos en el apartado de análisis se encuentran distribuidos entre dos máquinas virtuales. A una se le ha denominado `Openstack` y contiene todos los servicios referentes a Openstack - Módulos Swift, Keystone y Cliente Swift -. Cuenta con 2 procesadores, 4096 MB de memoria RAM y 50 GB de disco duro. La gran cantidad de recursos asignados es debido al coste que supone ejecutar Openstack. A la otra se le ha denominado `Kerberos` y en ella se encuentra el KDC. Posee un único procesador, 512 MB de memoria RAM y 8GB de disco duro. Como la carga de procesamiento de este equipo es más leve podemos limitar los recursos que se le proporcionan. En ambas se ha instalado un sistema operativo Ubuntu Server 12.04.3 LTS de 64 bits, que es el sistema que más compatibilidades ofrece con el software de Openstack.

Estos dos servidores se encuentran en una red local con direccionamiento 192.168.0.0/24, estando la primera dirección, la 192.168.0.1, reservada para el servidor `Kerberos` y la segunda, la 192.168.0.2, para el servidor `Openstack`. Tanto por comodidad de uso como por necesidad impuesta por el servicio Kerberos, del que se hablará más adelante, se ha desplegado mediante el software `Bind9` [74] una zona de resolución de nombres de dominio. Este software DNS se encuentra corriendo sobre el servidor `Kerberos`. Los nombres de dominio se han configurado de la siguiente forma:

- El nombre de dominio escogido es `openstack.um`.
- La IP 192.168.0.1 se corresponde con `kdc.openstack.um`. También los nombres `kadmin.openstack.um`, `ns.openstack.um` y `radius.openstack.um`.
- La IP 192.168.0.2 tiene asignado el nombre `keystone.openstack.um` y el nombre `swift.openstack.um`.
- Se ha configurado la resolución de nombres inversa para `keystone.openstack.um` y `kdc.openstack.um`.

El hecho de utilizar varios nombres para referenciar un mismo equipo puede parecer redundante, pero es importante recordar que en un escenario menos simplificado cada uno de estos servicios podría encontrarse en un equipo diferente, por lo que se trata de un buen hábito incluir todas estas posibilidades en la configuración del DNS. Los ficheros de configuración pueden ser encontrados en el anexo *'Configuración del servidor de nombres de dominio'*.

Como punto de inicio para el software de Computación en la Nube se ha tomado el código de Keystone ampliado por la Universidad de Kent con colaboración de la Universidad de Murcia [75]. El por qué utilizar esta versión y no la original es debido a la presencia de un mecanismo de autenticación mediante ABFAB/Moonshot que sirve como base a la hora de estudiar la integración de la GSS-API en Openstack.

Como software con el que hacer pruebas de autenticación y acceso a servicios se ha empleado el cliente de acceso al servicio Swift, también mejorado por la Universidad de Kent para el soporte de su mecanismo ABFAB/Moonshot [76].

Para realizar el despliegue han sido utilizadas las guías publicadas por la Universidad de Kent [77] [78]. Las guías por sí solas no son lo suficientemente completas y se requiere tener en cuenta algunos puntos adicionales:

- En la sección 2.1. se hace referencia a una forma de activar el módulo Swift que se encuentra actualmente desactivada, en las nuevas versiones es necesario sustituirlo por:

```
enable_service s-proxy s-object s-container s-account
```

  Ademas, puede aparecer un error al ejecutar el script de instalación. A pesar de esto, la instalación concluye con éxito y el error no afecta al desarrollo de los demás pasos.

- El apartado 2.2. está incompleto ya que no explica cuál es el contenido del fichero al que apunta. Esta descripción viene adjunta en el punto 8, pero a su vez resulta confusa ya que no especifica claramente que los campos `any_uid_project_ID` y `any_uid_role_ID` deben ser sustituidos por el identificador de un proyecto y un rol en particular de nuestros despliegue de Keystone.

- El punto 3.2 parece estar obsoleto y no ser soportado en la última versión de su software.

- Por último, la sección 5 es redundante y se puede omitir.

Tras superar todos estos contratiempos se debe contar con una versión operativa de Openstack con los módulos Keystone y Swift activos. Junto a estos también se activan los demás módulos como Nova, o Glance, los cuales pueden ser desactivados si se experimentan problemas por falta de recursos.

A continuación, se requiere la presencia de un entorno de autenticación Kerberos. En este caso, se ha desplegado el software presente en los repositorios de paquetes de Ubuntu y se ha configurado un *realm* Kerberos con el nombre de `OPENSTACK.UM`. Como ya se ha mencionado con anterioridad, tanto el AS como el TGS son ejecutados en la máquina `Kerberos`. El resumen del despliegue de este servicio está expuesto en el anexo 'Guión de instalación de Kerberos'.

En caso de que el cliente no realice una configuración automática basándose en el nombre de dominio sería necesario sustituir el contenido del fichero */etc/krb5.conf* por el que aparece en el anexo 'Guión de instalación de Kerberos'.

Finalmente, se despliega el servicio RADIUS que es esencial para las autenticación mediante mecanismos que hagan uso de ABFAB/Moonshot, aunque no es necesario si solo se desea probar el mecanismo de KerberosBasic. Para su despliegue se ha hecho uso de la guía que se adjunta en el anexo '*Guión de instalación de FreeRADIUS*'

RADIUS es el encargado de contactar con el IdP, que autenticará al usuario y transmitirá los atributos necesarios hasta el Keystone. Como este escenario está limitado a un nivel de pruebas, en lugar de obtener una sentencia de autenticación SAML mediante el IdP, será el propio RADIUS el encargado de generarla. Para que esto sea posible, es necesario configurar el servidor RADIUS para que introduzca la información adicional necesaria en las respuestas de autenticación. En particular, dicha información se trata de una sentencia SAML de autenticación idéntica a una que podría haber sido emitida por un IdP, en un momento concreto. De esta forma, Moonshot obtiene una respuesta de autenticación tal y como esperaba, sin tener que desplegar un IdP en el escenario de pruebas.

5.2. Implementación del mecanismo KerberosBasic

En esta sección explicaremos cómo se ha implementado el mecanismo de autenticación KerberosBasic en el módulo Keystone. Además, para poder hacer pruebas de uso se modificará un cliente Swift ya existente para soportar este nuevo mecanismo y poder así integrar la autenticación con Kerberos en un escenario de uso completo.

Partiremos de la base de que el usuario ya ha obtenido el ticket TGT. Este podría estar integrado con el inicio de sesión en el sistema o hacerse de forma manual mediante un cliente Kerberos. El mecanismo escogido nos es indiferente.

A continuación, explicaremos en los apartados siguientes las modificaciones necesarias en el cliente y en el servidor para ser adaptados al uso del nuevo mecanismo.

5.2.1. Modificaciones previas del servidor Keystone

Para que Keystone reconozca la presencia del mecanismo KerberosBasic es necesario hacer algunas modificaciones en los ficheros de configuración. En primer lugar, cuando Keystone recibe un mensaje de solicitud de autenticación como los visto en las figuras 15, 21 y 23 - entre otras -, consultará si dispone de algún *plugin* de autenticación denominado `KerberosBasic`. Para ello, debe estar presente un fichero con la implementación del mecanismo y además existir una entrada en el fichero de configuración de Keystone,

que, por defecto se encuentra en `/etc/keystone/keystone.conf`, que contenga una referencia a dicho mecanismo bajo la sección `[AUTH]`. En nuestro proyecto dicha entrada refleja lo siguiente:

```
kerberosBasic = keystone.auth.plugins.kerberosBasic.Kerberos
```

Lo que representa esta entrada es que el *plugin* denominado `kerberosBasic` se encuentra dentro de la ruta `/keystone/auth/plugins/kerberosBasic.py` y dentro de ese fichero, en la clase definida como *Kerberos*. Más adelante se describirán los detalles de implementación de ese fichero.

Junto a lo anterior, el servicio Keystone debe tener además configurado cuál es su FQDN y el nombre de registro empleado en el KDC. La combinación `Servicio@FQDN` será lo que emplee el mecanismo como su identificador al invocar a la GSS-API. Para automatizar esta configuración debemos añadir la siguiente entrada, bajo la misma sección del fichero de configuración mencionado anteriormente:

```
KerberosServiceHost = {NombreEscogido}@{FQDN}
```

Como ejemplo, en nuestro escenario quedaría de la siguiente forma:

```
KerberosServiceHost = keystone@keystone.openstack.um
```

Por último, para que el servicio keystone pueda reconocer dicha entrada como una variable del entorno Keystone es necesario modificar el fichero `keystone/common/config.py` y añadir a la variable `FILE_OPTIONS` la siguiente entrada:

```
cfg.StrOpt('KerberosServiceHost', default=None),
```

Con esto ya tenemos todos los cambios necesarios en la configuración de Keystone.

5.2.2. Código del mecanismo KerberosBasic

En esta sección se va a explicar el funcionamiento y la implementación del mecanismos de autenticación KerberosBasic. Puede consultar el código fuente en el anexo *Código fuente de KerberosBasic*.

En este código fuente desarrollado en el lenguaje de programación python aparece la clase `KerberosException` que nos sirve para el lanzamiento de excepciones relacionadas con el problemas de ejecución en la GSS-API. También está presente la clase `Kerberos` que hereda de la clase `AuthMethodHandler` y en cuyo interior se desarrolla toda la lógica del mecanismo de autenticación. La herencia sobre la clase AuthMethodHandler es un requisito para la creación de cualquier mecanismo de autenticación de Keystone.

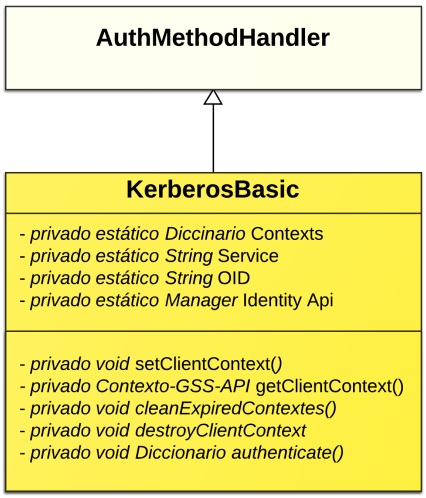


Figura 26: Diagrama UML del método KerberosBasic

Los métodos y atributos de los que hace uso esta clase son los siguientes:

- Atributos estáticos:
  - `Contexts`: se trata de un diccionario donde se almacenan los contextos generados por la GSS-API indexados por el UUID de la negociación.
  - `Service`: este atributo carga la variable de configuración de Keystone KerberosServiceHost, que contiene el nombre utilizado por el servicio Keystone para registrarse en el KDC de Kerberos. Este atributo será utilizado en el establecimiento del contexto GSS-API.
  - `OID`: identificador OID del mecanismo Kerberos. Este identificador es proporcionado a la GSS-API a la hora de establecer el contexto para indicar que queremos usar Kerberos como mecanismos de autenticación.
  - `Identity_api`: referencia a la clase gestora de las identidades de usuario en el módulo Keystone. Con ella podemos obtener datos del usuario después de que haya sido autenticado.
- Métodos:

- `Mecanismos de gestión del contexto:` encargados de la gestión - almacenamiento, obtención y eliminación -, de los contextos de seguridad GSS-API.
- `authenticate (self, context, auth_payload, user_context )`: este es el método central de mecanismo de autenticación. Cada vez que Keystone recibe una solicitud de autenticación invoca al método `authenticate` del mecanismo de autenticación correspondiente. A continuación se detalla su funcionamiento.

Este método `authenticate` recibe como parámetros las variables `context`, `auth_payload` y `user_context`. En `context` se almacena el contexto de la solicitud que el cliente ha enviado como petición de autenticación al módulo Keystone; `auth_payload` contiene el cuerpo de dicha solicitud y `user_context` se utiliza para almacenar la información sobre el usuario una vez ha completado el proceso, de forma que Keystone pueda saber qué usuario ha sido autenticado.

Esta función puede acabar de tres maneras diferentes:

1. **El proceso de autenticación ha sido completado.** Se indica almacenando en la variable `user_context['user_id']` el ID del usuario autenticando y retornando el valor `None`:

```
# Buscamos el ID del usuario asociado a la negociacion
userName =  pymoonshot.authGSSServerUserName(context['context'])
usuario = Kerberos.identity_api.get_user_by_name(userName, '
 default')
user_context['user_id'] = usuario['id']
return None
```

2. **Se necesitan más datos en el proceso de autenticación**. Ocurre, por ejemplo, cuando la GSS-API requiere enviar o recibir más tokens GSS-API antes de dar por finalizada la autenticación. En este caso se devuelve un diccionario con la información que se quiere transmitir en el mensaje HTTP de respuesta. Por ejemplo, el siguiente código:

```
#Establecemos un ID de sesion
return {'cid':uuid.uuid4().hex, 'negotiation':''}
```

Daría un resultado similar al del mensaje siguiente:

```
{
  "error": {
    "message": "Additional authentications steps required.",
    "code": 401,
    "identity": {
      "methods": ["kerberosBasic"],
      "kerberosBasic": {
        "negotiation": "",
        "cid": "4391bb9ef3d74af199b8fec805bb7fd5"}
        },
    "title": "Unauthorized"
  }
}
```

3. **Que haya ocurrido un error durante la negociación**. Los tipos de errores que pueden surgir son variados y de distinta naturaleza. Podría ser que el cliente esté usando credenciales inválidas y la GSS-API diera la autenticación por fallida, o que el cuerpo de la petición HTTP no se adapte al formato esperado. Para indicar estas situaciones se utiliza el lanzamiento de excepciones. En el código podemos encontrar ejemplos como los siguientes:

```
# Por error de formato
raise exception.ValidationError(attribute='cid', target=
 auth_payload)
#Por fallo en la GSS-API
raise KerberosException('moonshot.authGSSServerInit returned %d:'
  %state)
#Excepción que captura todas las demás y produce una genérica
except (Exception), err:
    LOG.exception(err)
    raise exception.Unauthorized('Unauthorized user')
```

Ahora que sabemos de qué forma puede finalizar la ejecución de la negociación en el Keystone, vamos a explicar en detalle cual es la finalidad del código que hemos implementado en el método `authenticate`:

El primer paso consiste en la **identificación de la negociación**. El código comienza buscando el atributo `cid` en el cuerpo de la solicitud HTTP mediante la variable `auth_payload`. Si es vacío, es posible que se trate del primer mensaje de la negociación (Mensaje tipo 1, ver figura 15). Para comprobarlo buscamos el atributo `phase` y comprobamos que su valor sea `discovery`; en ese caso contestamos con un mensaje similar al de la figura 16.

```
# Recogemos el CID
if 'cid' in auth_payload  and auth_payload['cid'] is not None:
  cid = uuid.UUID(auth_payload['cid']).hex

# Si es vacío, puede ser que se trate del mensaje inicial
elif 'phase' in auth_payload and
 auth_payload.get('phase') == 'discovery':
  # Establecemos un ID de sesion
  return {'cid':uuid.uuid4().hex, 'negotiation':''}
```

El siguiente paso es comprobar la fase actual del proceso. Para ello, miramos el valor del atributo `phase`. Llegados a este punto puede tener dos posibles valores: `negotiate` o `complete`.

Si el valor contenido es `negotiate` nos encontramos en el proceso de negociación con la GSS-API. Lo primero que hacemos es comprobar que exista el atributo `Negotiation` en el mensaje recibido; y si no existe lanzamos una excepción indicando el fallo en el formato del mensaje. El siguiente paso es recoger el contexto GSS-API de la negociación, que como ya hemos mencionado anteriormente se realiza mediante la función `getClientContext(cid)`.

```
#Cogemos el token gss-api
negotiation = auth_payload.get('negotiation')
# En otro caso recogemos el contexto GSS-API asociado
contextGSSAPI = self.getClientContext(cid)
```

Puede ser que el contexto sea nulo en primera instancia si nos encontramos en el primer mensaje de la fase de negociación que explicamos en la sección 4.2.2. Para crear un

contexto GSS-API haremos uso del método `AuthGSSServerInit` de la librería PyMoonshot [3]. A este método se le pasan como atributos un identificador del servicio formado por el nombre del servicio y el FQDN utilizados para registrar el servicio en el KDC de Kerberos; y el OID del mecanismo que queremos emplear en ese contexto GSS-API. Estos valores se encuentran en los atributos `Kerberos.service` y `Kerberos.oid` respectivamente. Como resultado la función devuelve un código de estado y un contexto. El código de estado debe tener un valor mayor o igual que cero para indicar que no ha ocurrido ningún error en la creación del contexto.

```
try:
# Si no hay un contexto significa que es la primera llamada a la GSS-
    API
  if contextGSSAPI is None:
    #Inicializamos el contexto
    contextGSSAPI = {}
    # Primera invocación a la GSS-API que nos construye un contexto
    contextGSSAPI['state'], contextGSSAPI['context'] = pymoonshot.
   authGSSServerInit(Kerberos.service, Kerberos.oid)
    # Comprobamos si ha habido algún error mirando el estado
    if contextGSSAPI['state'] < 0:
      raise KerberosException(
      'authGSSServerInit returned %d:' % state)
```

Una vez tengamos el contexto GSS-API - ya sea porque lo acabamos de inicializar, o porque ya estaba creado en algún momento anterior y lo hemos recuperado mediante el método `getClientContext`-, el siguiente paso es entregarle a la GSS-API el token GSS-API recibido por parte del cliente. Para esto se hace uso de la función `authGSSServerStep` de la librería PyMoonshot.

Esta función recibe como parámetros el contexto y el token GSS-API y emite como resultado un estado similar al devuelto por la función `AuthGSSServerInit`. Tras almacenar el contexto para tenerlo disponible en la siguiente invocación del cliente. Para acabar, lo único que nos resta es recuperar el token GSS-API que debemos retornar al cliente. Esto se consigue invocando al método `authGSSServerResponse` de la misma librería, al que se le pasa como parámetro el contexto GSS-API.

Una vez obtenido el token, lo insertamos como campo en el objecto `resp` que como recordaremos contiene un campo con el atributo `cid` con el UUID de la negociación y

[3]PyMoonshot: librería publicada por Vincent Giersch en su cuenta de Github [79]. Su finalidad es la de ofrecer un wrapper de alto nivel en lenguaje Python de los mecanismos de la librería GSS-API que a su vez se encuentra implementada sobre el lenguaje C.

otro campo llamado `negotiation` que contiene dicho token. Finalmente, acabamos este paso devolviendo `resp` como resultado de la ejecución del método `authenticate`.

```
# Aquí ya tenemos un contexto GSS-API construido
# Le pasamos el token GSS-API recibido para que lo procese
contextGSSAPI['state'] = pymoonshot.authGSSServerStep(
  contextGSSAPI['context'], negotiation)

# Almacenamos el contexto GSS-API para una consulta posterior
self.setClientContext(cid, contextGSSAPI)
resp['negotiation'] = pymoonshot.authGSSServerResponse(
  contextGSSAPI['context'])

# Devolvemos el mensaje construido con el token GSS-API y el UUID
return resp
```

Como alternativa a lo anterior, podríamos encontrarnos en el caso de tener `complete` como valor del atributo phase. En esta situación, comenzaremos buscando el contexto GSS-API asociado a la negociación con el método `getClientContext(cid)`. En caso de no existir - por ejemplo porque alguien haya intentado saltarse el proceso de autenticación indicando directamente que su autenticación ha terminado -, se lanzaría una excepción que finalizaría la negociación enviando un mensaje de error.

El siguiente paso después de obtener con éxito el contexto GSS-API es preguntar por el usuario asociado a la GSS-API. Para esto podemos hacer uso de la función `authGSSServerUserName()` de la librería PyMoonshot. Con el nombre podemos comprobar si pertenece a nuestro *realm* Kerberos o se trata de un usuario federado mirando la parte final. Si se trata de un usuario federado, se sustituye por un usuario común que hemos denominado `federado`. Con esto se soluciona el problema del mapeo de usuarios federados del que hablamos en la sección *4.3*.

Ya solo nos resta obtener el identificador en OpenStack de dicho usuario. Esto es fácil de realizar mediante el método `get_user_by_name(userName)` del atributo `identity_api` que, como recordamos, es un módulo encargado de la gestión de usuarios. De el objecto usuario obtenemos su identificador que se almacena en el atributo `user_context[‘user_id’]` para indicar a Keystone que ese ha sido el usuario autenticado. Finalmente, devolvemos `None` que es la forma de indicar a Keystone que se ha autenticado con éxito al usuario.

```
# Buscamos el ID del usuario asociado a la negociacion
userName =  pymoonshot.authGSSServerUserName(
  contextGSSAPI['context'])

# Mapeo para usuarios autenticados con Kerberos Cross Realm
if not userName.endswith('@OPENSTACK.UM'):
  userName = 'federado'

usuario = Kerberos.identity_api.get_user_by_name(userName,'default')
user_context['user_id'] = usuario['id']
return None
```

Con esto está completa la funcionalidad del servidor Keystone para la autenticación mediante Kerberos. Ahora sólo resta adaptar el cliente Swift para que implemente la parte cliente y poder ejecutar un escenario completo de acceso a un servicio mediante autenticación con Kerberos.

5.3. Adaptación del cliente Swift al mecanismo KerberosBasic

Para poder realizar una ejecución completa del escenario es necesario adaptar el cliente Swift para que use el mecanismo KerberosBasic. El primer paso para ello es que el usuario sea capaz de indicar que quiere realizar la autenticación mediante este mecanismo.

Para indicarlo haremos uso del flag `--kerberos` a la hora de realizar la consulta. Activar esta opción requiere modificar el fichero `bin\swift` en el directorio swiftcliente instalado en el despliegue del entorno Openstack - que en general se encuentra en `/opt/stak/python-swiftclient` -, y añadir en la función principal lo siguiente:

```
  parser.add_option('--kerberos', dest='kerberos', action="store_true",
                    help='Use Kerberos for authentication.')
```

La forma con la que la interfaz le indica al cliente swift qué mecanismo de autenticación debe utilizar - 1.0, 2.0, Moonshot o Kerberos -, es mediante la variable `options.auth_version`. Así pues, en caso de que el usuario marque la opción `--kerberos` anterior, debemos asegurarnos de que se le asigna un valor especial en esta variable que permita luego informar al módulo de autenticación de la nueva opción. Para ello, en la función `parse_args` del mismo fichero anterior se añade:

```
    if options.kerberos:
        options.auth_version='Kerberos'
```

El punto de entrada para el módulo de autenticación se encuentra en la función `get_auth` de la clase `Connection` en el fichero `swifclient/client.py`. Al ejecutarse comprueba el valor de la variable `options.auth_version` de la que hablábamos anteriormente, e invoca al mecanismo de autenticación particular en cada caso. Para adaptar la función al uso de nuestro mecanismo es suficente con añadir en la secuencia de `IFs` que la componen una nueva entrada:

```
import swiftclient.contrib.kerberos as kerberos

def get_auth(url, user, key, snet=False, tenant_name=None, auth_version
   ="1.0", realm=None):
    if auth_version in ["1.0", "1"]:
        return _get_auth_v1_0(url, user, key, snet)
    elif auth_version in ["2.0", "2"]:
        if not tenant_name and ':' in user:
            (tenant_name, user) = user.split(':')
        if not tenant_name:
            raise ClientException('No tenant specified')
        return _get_auth_v2_0(url, user, tenant_name, key, snet)
    elif auth_version in ["F", "federated"]:
        return _get_auth_federated(url, realm, tenant_name)
    elif auth_version == "kerberos":
        return _get_auth_kerberos(url,tenant_name)
    else:
        raise ClientException('Unknown auth_version %s specified.'
                              % auth_version)
```

```
def _get_auth_kerberos(url, tenant_name):
    requestPool = urllib3.PoolManager()
    kerberosObject = kerberos.KerberosNegotiation(url, requestPool,
   tenant_name)
    (url, token) = kerberosObject.negotiation()
    return url, token
```

En nuestro caso, hemos hecho uso de la función `_get_auth_kerberos` que también ha sido incluida. Con su ayuda será preparado lo necesario para invocar finalmente al

mecanismo de autenticación KerberosBasic. La preparación consiste en obtener un gestor de conexiones HTTP - proporcionado por la librería para conexiones HTTP `urllib3` -, que será usado por el mecanismo para establecer la conexión con el servidor, construir un objecto `KerberosNegotiation`. Para ello se pasan como parámetros la dirección del keystone almacenada en el parámetro `url`, el gestor de conexiones HTTP y, opcionalmente, el nombre del tenant al que se quiere limitar el acceso del token Keystone. Por último, se realiza la invocación a su método `negotiation`. Con esto se lanzará la autenticación, y al finalizar se obtendrá la url de acceso al servicio Swift y el token Keystone.

Para implementar la clase KerberosNegotiation hay que crear el fichero `swiftclient/contrib/keystone.py` e insertarle el código fuente que puede consultarse en el anexo *Código de KerberosBasic en el cliente Swift*.

Como se mencionaba anteriormente, al ejecutar el mecanismo KerberosBasic la ejecución comienza por el método `negotiation`. En él, la lógica de aplicación se puede dividir en los siguientes pasos:

1. **Envío y recepción de los mensajes `KerberosBasic Discovery`**: Para ello se hace uso del gestor de conexiones HTTP.

```
## Envío del mensaje KerberosBasic (Discovery Request)
  body = '{"auth": {"identity": {"methods": ["kerberosBasic"], "
 kerberosBasic" : {"phase":"discovery"}}}}'
  headers = {"Content-type": "application/json"}
  resp  = json.loads(self.requestPool.urlopen('POST', self.
 keystoneEndpoint, body = body, headers = headers).data)

## Recepción del mensaje KerberosBasic(Discovery Response)
  discoveryResponse = resp['error']['identity']['kerberosBasic']
  self.idpResponse = {'cid':discoveryResponse['cid'],'negotiation
 ':''}
```

2. **Creación del contexto de seguridad GSS-API y obtención del GSS-TOKEN**: Al igual que en el servidor, para el uso de la librería GSS-API se emplea la librería PyMoonshot. En particular, para la creación del contexto de seguridad GSS-API se invoca al método `authGSSClientInit` al que se le pasa como parámetros el código del OID del mecanismo Kerberos y el nombre del principal del servicio Keystone obtenido del mensaje anterior.

```
## Establecimiento del contexto gss-api
  result, self.context = pymoonshot.authGSSClientInit(
 discoveryResponse['service'],
       pymoonshot.GSS_C_MUTUAL_FLAG | pymoonshot.
 GSS_C_SEQUENCE_FLAG, self.mechanism)
```

3. **Autenticación mediante GSS-API**: Mientras la GSS-API no muestre el estado de completado se intercambian tokens GSS-API con Keystone. Para ello se hace uso de la función `authGSSClientStep` encargada de procesar el token GSS-API - aunque en el primer se encontrará vacío -, y se obtiene el token de respuesta mediante la función `authGSSClientResponse`. Tras obtenerlo, se envía haciendo uso de la función auxiliar `negotiationRequest`, que se limita a crear un mensaje HTTP con el formato del mensaje KerberosBasic Negotiate y enviarlo al servicio Keystone.

```
 # Envío de los mensajes KerberosBasic(negotiate)
      while negotiation != pymoonshot.AUTH_GSS_COMPLETE:
          negotiation = self.negotiationStep()

def negotiationStep(self):
result = pymoonshot.authGSSClientStep(self.context, self.
 idpResponse['negotiation'])
idpNegotiation = pymoonshot.authGSSClientResponse(self.context);
if idpNegotiation is not None:
  server_resp = self.negotiationRequest(idpNegotiation, self.
 idpResponse.get("cid", None));
  self.idpResponse = {"negotiation": server_resp["error"]
   ["identity"]["kerberosBasic"]["negotiation"],
   "cid":server_resp["error"]["identity"]
  ["kerberosBasic"].get("cid", None)}
return result
```

4. **Fin de la autenticación y obtención del token Keystone**: una vez que la función result anterior indique que se ha completado la autenticación se puede solicitar el token Keystone a sabiendas de que la autenticación ha sido exitosa. Para solicitarlo se construye un mensaje `KerberosBasic Complete` y se envía con ayuda del gestor de conexiones HTTP. Del mensaje de respuesta se obtiene tanto

el token de autenticación de Keystone, como la URL del servicio Swift. Retornando estos dos valores el cliente swift es capaz, por si mismo, de continuar con el acceso al servicio.

```
url = None
## Devolvemos el token id y la url del servicio
if "token" in self.idpResponse and "catalog" in self.idpResponse[
 "token"]:
  for cat in self.idpResponse["token"]["catalog"]:
    if cat["type"] == "object-store":
      for endp in cat["endpoints"]:
        if endp["interface"] == "public":
          url = endp["url"]
  if url == None:
    url = self.keystoneEndpoint

  return url, resp.getheader("x-subject-token")
```

5.4. Prueba del mecanismo KerberosBasic

En esta sección vamos a mostrar un escenario en el que el cliente Swift modificado realiza una autenticación mediante el mecanismo KerberosBasic. Comenzaremos obteniendo el ticket TGT necesario para la obtención de acceso a servicios kerberizados. Si no se encuentra integrado con el inicio de sesión, lo podemos obtener mediante el cliente Kerberos con el comando `kinit` en una terminal del sistema.

```
$ kinit
Password for victor@OPENSTACK.UM:
```

Con el comando `klist` podemos ver tanto los tickets actuales de los disponemos como información sobre su periodo de validez. Por ejemplo, el ticket TGT obteniendo en el paso anterior aparece bajo el nombre del *principal* `krbtgt/OPENSTACK.UM@OPENSTACK.UM` y con un periodo de validez entre las 15:59:57 del 06/07/14 hasta la misma hora del día 07/07/14.

```
$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: victor@OPENSTACK.UM

Valid starting     Expires            Service principal
06/07/14 15:59:57  07/07/14 15:59:57  krbtgt/OPENSTACK.UM@OPENSTACK.UM
  renew until 06/07/14 15:59:57
```

A continuación ejecutamos el cliente Swift indicando que queremos recibir la lista de contenedores del usuario como ejemplo de acceso al servicio. Para ello incluimos los parámetros `--kerberos` para indicar que queremos hacer la autenticación mediante el método `KerberosBasic` junto con la opción `-A` indicamos la dirección del endpoint donde Keystone espera recibir las peticiones.

```
$ swift --kerberos -A http://keystone.openstack.um:5000/v3/auth/tokens
   list
```

El mecanismo KerberosBasic se encargará de autenticar al usuario sin que este tenga que volver a introducir ningún credencial. Finalmente, el usuario recibe directamente la resolución de su petición sin que necesite realizar pasos adicionales. El primer mensaje mostrado por el cliente Swift al usuario, es información adicional indicando que se ha completado el proceso de autenticación mediante Kerberos con un determinado principal. El resto de entradas son el resultado de la consulta realizada:

```
Authentication successful using "victor@OPENSTACK.UM" kerberos identity
Contenedor2
Contenedor3
contenedor
```

Como se puede comprobar, en la caché de tickets ahora aparece el ticket kerberos ST de acceso al principal Keystone - `keystone/keystone.openstack.um@OPENSTACK.UM` - obtenido de forma transparente al usuario:

```
$ klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: victor@OPENSTACK.UM

Valid starting     Expires            Service principal
06/07/14 15:59:57  07/07/14 15:59:57  krbtgt/OPENSTACK.UM@OPENSTACK.UM
  renew until 06/07/14 15:59:57
06/07/14 16:00:12  07/07/14 15:59:57  keystone/keystone.openstack.
    um@OPENSTACK.UM
  renew until 06/07/14 15:59:57
```

Con ayuda del programa Wireshark [80] para análisis de tráfico de red se puede observar como los mensajes intercambiados son los descritos en la sección de análisis:

Source Port	Destination Port	Protocol	Info
33433	kerberos	KRB5	AS-REQ
kerberos	33433	KRB5	AS-REP
56934	keystone	HTTP	POST /v3/auth/tokens/ HTTP/1.1 (application/json)
keystone	56934	HTTP	HTTP/1.1 401 Unauthorized (application/json)
32783	kerberos	KRB5	TGS-REQ
kerberos	32783	KRB5	TGS-REP
56934	keystone	HTTP	POST /v3/auth/tokens/ HTTP/1.1 (application/json)
keystone	56934	HTTP	HTTP/1.1 401 Unauthorized (application/json)
56934	keystone	HTTP	POST /v3/auth/tokens/ HTTP/1.1 (application/json)
keystone	56934	HTTP	[TCP Previous segment not captured]
59940	swift	HTTP	GET /v1/AUTH_0abaeb222f024f8cbf3afcae43d823c6
swift	59940	HTTP	HTTP/1.1 200 OK (application/json)

Figura 27: Tráfico generado en el acceso al servicio.

En ella aparecen los siguientes mensajes por orden descendente:

1. **Solicitud `AS_REQ`**: para autenticar al usuario en el entorno Kerberos. En su interior se indica el nombre del *principal* del usuario - `victor` -, su *realm* - `OPENSTACK.UM` -, el principal del TGS - krbtgt/OPENSTACK.UM -, y la fecha de validez de la sesión.

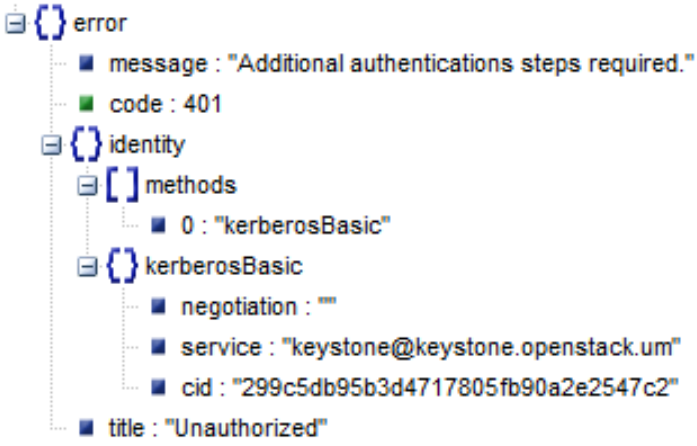
```
Kerberos AS-REQ
  Pvno: 5
  MSG Type: AS-REQ (10)
⊞ padata: Unknown:149
⊟ KDC_REQ_BODY
    Padding: 0
  ⊞ KDCOptions: 50000010 (Forwardable, Proxiable, Renewable OK)
  ⊞ Client Name (Principal): victor
    Realm: OPENSTACK.UM
  ⊞ Server Name (Service and Instance): krbtgt/OPENSTACK.UM
    from: 2014-07-06 14:00:06 (UTC)
    till: 2014-07-07 14:00:06 (UTC)
    Nonce: 159727408
  ⊞ Encryption Types: aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96 des3-cbc-sha1 rc4-hmac
```

2. **Respuesta `AS_REP`**: contiene el TGT para el usuario. El mensaje se compone de una parte con información pública - *realm* y nombre del TGS -, y una parte cifrada (en rojo en la figura 2) que sólo podrá ser obtenida por el usuario gracias a que conoce la clave que ha utilizado el AS para cifrarlo.

```
Kerberos AS-REP
  Pvno: 5
  MSG Type: AS-REP (11)
⊞ padata: PA-ENCTYPE-INFO2
  Client Realm: OPENSTACK.UM
⊞ Client Name (Principal): victor
⊟ Ticket
    Tkt-vno: 5
    Realm: OPENSTACK.UM
  ⊞ Server Name (Service and Instance): krbtgt/OPENSTACK.UM
  ⊟ enc-part aes256-cts-hmac-sha1-96
      Encryption type: aes256-cts-hmac-sha1-96 (18)
      Kvno: 1
      enc-part: 506b2bbb8eb258413e54d89bae817156f266737c0bd3e12f...
⊞ enc-part aes256-cts-hmac-sha1-96
```

3. **Solicitud KerberosBasic Discovery**: mensaje similar al descrito por el modelo teórico.

4. **Respuesta KerberosBasic Discovery**: contiene el identificador de sesión que el usuario deberá utilizar de ahora en adelante - contenido en `cid` -, y el *principal* del servicio Keystone - presente en `service` -.



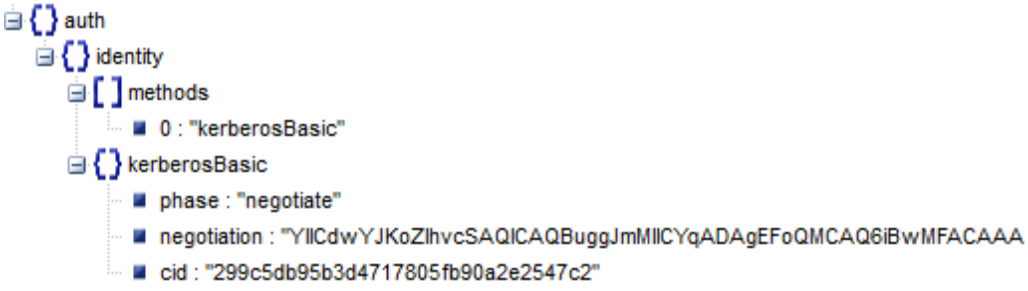
5. **Solicitud TGS_REQ**: con este mensaje se solicita la obtención del ticket ST para el usuario. Su estructura es muy similar a la del mensaje `AS_REQ`, siendo el cambio más destacable el *principal* para el que se solicita el ticket - en este caso `keystone/keystone.openstack.um`-.

```
Kerberos TGS-REQ
  Pvno: 5
  MSG Type: TGS-REQ (12)
⊞ padata: PA-TGS-REQ
⊟ KDC_REQ_BODY
    Padding: 0
  ⊞ KDCOptions: 50810000 (Forwardable, Proxiable, Renewable, Canonicalize)
    Realm: OPENSTACK.UM
  ⊞ Server Name (Service and Host): keystone/keystone.openstack.um
    till: 2014-07-07 13:59:57 (UTC)
    Nonce: 1404655209
  ⊞ Encryption Types: aes256-cts-hmac-sha1-96 aes128-cts-hmac-sha1-96 des3-cbc-sha1 rc4-hmac
```

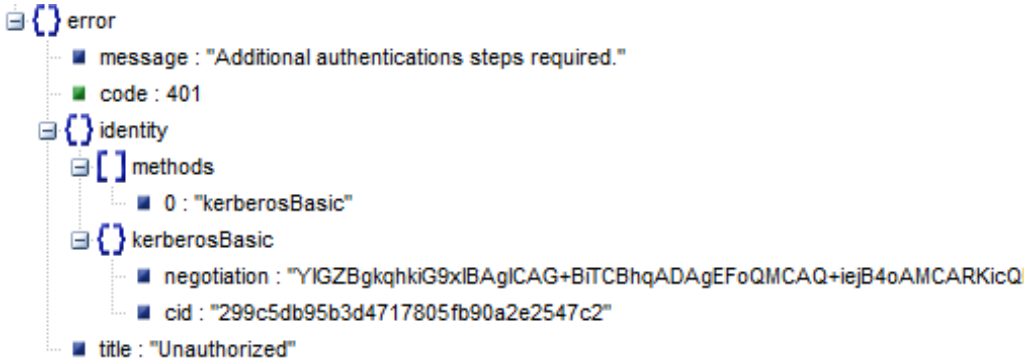
6. **Respuesta TGS_REP**: de la misma manera, su respuesta es similar al mensaje `AS_REP` anterior. De nuevo, el ticket tiene unos datos públicos; mientras que el ticket ST va protegido mediante encriptación. La principal diferencia es que en este caso el usuario no conoce la clave de cifrado, por lo que no podrá extraer ni, por tanto, manipular el ticket para falsificarlo.

```
Kerberos TGS-REP
  Pvno: 5
  MSG Type: TGS-REP (13)
  Client Realm: OPENSTACK.UM
⊞ Client Name (Principal): victor
⊟ Ticket
    Tkt-vno: 5
    Realm: OPENSTACK.UM
  ⊞ Server Name (Service and Host): keystone/keystone.openstack.um
  ⊟ enc-part aes256-cts-hmac-sha1-96
      Encryption type: aes256-cts-hmac-sha1-96 (18)
      Kvno: 2
      enc-part: 33cac0caa3808b606252f82c877037b4f4cd9c83150fdeda...
⊟ enc-part aes256-cts-hmac-sha1-96
    Encryption type: aes256-cts-hmac-sha1-96 (18)
    enc-part: c1704feecdf2478597aa68cf59733258bc8bd1f1c3024da1...
```

7. **Solicitud KerberosBasic Negotiate**: con la información anterior, se genera un token GSS-API que se inserta en el campo `negotiation`. Junto a él, también se indica el identificador de sesión para que el Keystone pueda distinguirlo como parte de la misma negociación.

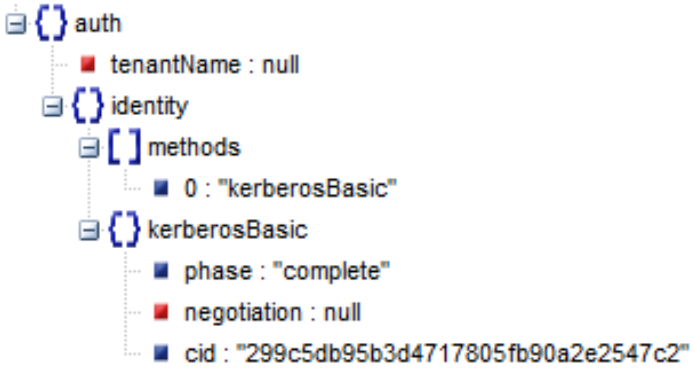


8. **Respuesta KerberosBasic Negotiate**: en respuesta, Keystone contesta con el token GSS-API generado por la implementación local de dicha API.



9. **Solicitud KerberosBasic Complete**: tras autenticar al servicio Keystone, el cliente envía este mensaje para solicitar el token Keystone. Si se deja en el campo

`tenantName` vacío - como ocurre si no se activa el *flag* -T en la ejecución del cliente -, se obtendrá un *unscoped token*; mientras que si se especifica un nombre, se obtendrá un *scoped token* con alcance limitado a dicho *tenant*.



10. **Respuesta KerberosBasic Complete**: el mensaje recibido ya no forma parte del mecanismo KerberosBasic, sino que se trata de la respuesta genérica que presenta Keystone cuando el usuario ha logrado autenticarse - independientemente del mecanismo empleado para ello -, y que contiene el token Keystone.

11. **Solicitud de un servicio en Swift**: se trata de un mensaje HTTP GET. La URL de la petición especifica a qué mecanismo de la API Swift quiere acceder, y en las cabeceras se incluye el token obtenido en el paso anterior.

12. **Respuesta del servicio Swift**: un mensaje HTTP 200 con la información del servicio. En las cabeceras `X-Timestamp`, `X-Account-Bytes-Used`, `X-Account-Container-Count`, `X-Account-Object-Count` y `X-Trans-Id` se incluye información adicional sobre la operación.

```
HTTP/1.1 200 OK
Content-Length: 146
Accept-Ranges: bytes
X-Timestamp: 1396454616.66239
X-Account-Bytes-Used: 0
X-Account-Container-Count: 3
Content-Type: application/json; charset=utf-8
X-Account-Object-Count: 0
X-Trans-Id: txe1b6958e4f2a4d4c9c73f-0053b95675
Date: Sun, 06 Jul 2014 14:00:21 GMT

[
  {"count": 0, "bytes": 0, "name": "Contenedor2"},
  {"count": 0, "bytes": 0, "name": "Contenedor3"},
  {"count": 0, "bytes": 0, "name": "contenedor"}
]
```

5.5. Kerberos y el entorno Moonshot de OpenStack

En este apartado se analizará cómo funciona la solución aportada por la Universidad de Kent para dar soporte a ABFAB/Moonshot en Keystone y se realizará una breve prueba de concepto sobre cómo se podría incluir Kerberos como mecanismo adicional dentro de dicho interfaz. Se recuerda que dicho entorno está disponible ya que, durante el despliegue del escenario se seleccionó el módulo Keystone adaptado por esta universidad para el dar soporte a la arquitectura ABFAB/Moonshot.

En primer lugar, se explicará cómo se puede solicitar la autenticación por medio de los mecanismo de dicho entorno. Para ello, el cliente debe enviar un mensaje de solicitud de autenticación que tenga como cuerpo de la solicitud el siguiente mensaje:

```
{"auth": {
  "identity": {
    "methods": ["federated"],
    "federated": {
      "phase": "discovery"}}}
```

En el servicio Keystone se lanzará el mecanismo desarrollado por Kent, que se encargará de buscar todos los mecanismos disponible dentro del entorno Moonshot para la autenticación del usuario. Para ello, buscará en el módulo Keystone todos los servicios

registrados cuyo atributo `type` comience por '`idp.`'. A continuación, buscará en el fichero de configuración de Keystone la ruta a las clases que contienen la lógica de autenticación para cada mecanismo y las cargará en el sistema. Por último, contestará al usuario devolviendo la lista con los mecanismos anteriormente obtenida.

El usuario selecciona qué mecanismo desea utilizar mediante la interfaz que incluye la implementación en el cliente Swift. En su última versión están presentes los mecanismos `ABFAB`, `SAML`, y `Keystone`; pero parece que sólo se mantiene el soporte del primero. Cuando se selecciona uno de los mecanismos, enviará otra petición de autenticación con tipo de mecanismo `federated` y con la información del mecanismo seleccionado.

Gracias al uso de la GSS-API, existe una única implementación en el lado del cliente capaz de trabajar con cualquiera de los mecanismos de autenticación. También en el lado del servidor la implementación es casi totalmente independiente, siendo necesario únicamente modificar la lógica del mecanismo tras la autenticación vía GSS-API - por ejemplo, para recuperar información adicional del usuario desde su IdP -.

Tras esta breve explicación del funcionamiento del entorno se describe cómo se ha incluir el protocolo Kerberos para ser utilizado junto al resto de los mecanismos anteriores:

1. **Registro del servicio en Keystone**: como se ha explicado anteriormente, para que un mecanismo sea reconocido en este módulo es necesario que esté registrado como servicio. Para ello se hace uso de los siguientes comandos en una terminal del equipo servidor:

```
$ keystone service-create --name service-name \
 --type idp.kerberos \
 --description "Kerberos Identity Provider"

$ keystone endpoint-create --service-id {ServicioID} \
--publicurl {PublicURL} --internalurl  {internalURL} \
 --adminurl {adminURL}
```

2. **Creación del mecanismo Kerberos**: gracias a la independencia del mecanismo con el que se ha desarrollado el código del mecanismo ABFAB, este puede ser reconvertido fácilmente en un mecanismo Kerberos. Para ello, basta con realizar una copia del fichero `keystone\auth\federated\protocol\abfab.py` a `kerberos.py`. Tras esto modificamos el nombre de clase de `ABFAB` a `Kerberos` y sustitumos la función `request_auth` por la siguiente:

```
self.service = 'keystone@%s' %socket.gethostname()
self.oid =  '{1 2 840 113554 1 2 2}'

# Plugin steps
def request_auth(self, protocol_data):
    return {
        'mechanism': self.oid,
        'service_name': self.service
    }
```

Su función es la de generar un mensaje de respuesta para el cliente Swift, con los datos que debe proporcionar a su GSS-API local para realizar el proceso de autenticación con dicho mecanismo. En concreto, en él se encuentra contenido el identificador y el OID del servicio a utilizar.

3. **Postautenticación con Kerberos**: como ya hemos dicho, no todo el proceso es independiente del mecanismo. Tras realizar la autenticación con GSS-API se invoca a la función `validate`, encargada de recoger los atributos del usuario proporcionados por el IdP. Como en este caso no hay un IdP que los proporcione, se ha forzado a que se devuelva una información concreta que represente a una usuario ya existente en la base de datos - a modo de prueba -:

```
return 'test',{'portal_id':['test'],
'urn:oid:1.3.6.1.4.1.5923.1.1.1.7':['moonshot']},'2017-12-10T19
 :39:48Z'
```

En un escenario más completo, esos datos serían traídos por el KDC desde el IdP del usuario, e insertados en el interior del ticket Kerberos ST. Cuando el Keystone reciba este ticket Kerberos como parte del proceso de autenticación obtiene de él la información adicional sobre el usuario - p. ej. sus roles -. De esta forma se lograría un autentico acceso federado, cosa que no ocurre en esta prueba de concepto.

4. **Añadir mecanismo al fichero de configuración**: finalmente, ya sólo queda especificar el mecanismo en el fichero `keystone.conf` para que pueda ser ejecutado por Keystone. Para ello modificamos su contenido para que quede de la siguiente manera:

```
protocols =  abfab, kerberos
kerberos = keystone.auth.plugins.federated.protocol.kerberos.
 Kerberos
```

Tras estos pasos, se puede realizar la autenticación:

```
$ swift -F -A http://keystone.openstack.um:5000/v3 list
Please use one of the following services to authenticate you:
        { 0 }  service-name
        { 1 }  kerberos
        { 2 }  service-name
Enter the number corresponding to the service you want to use: 0
Authentication successful using "victor@OPENSTACK.UM" kerberos identity
   .
You have access to the following tenant(s)and domain(s):
        { 0 }  swifttenanttest1
Enter the number corresponding to the tenant you want to use: 0
Contenedor2
Contenedor3
contenedor
```

Es importante recordar que aunque se ha integrado Kerberos para trabajar dentro del entorno de mecanismos ABFAB/Moonshot que ha diseñado la Universidad de Kent, esto no significa que se haya integrado Kerberos dentro de la arquitectura ABFAB. La prueba de concepto anterior no permite la autenticación federada de usuarios, quedando esto como un posible trabajo posterior.

6. Conclusiones y vías futuras

Durante las diferentes secciones de este trabajo se ha presentado un mecanismo de autenticación de usuarios mediante la arquitectura Kerberos para los servicios del software de Computación en la Nube OpenStack. Este mecanismo delega la gestión del material criptográfico generado por Kerberos para la autenticación tanto del usuario como del servicio, a la *Generic Security Services Application Program Interface* (GSS-API). Gracias al uso de esta interfaz se elimina la dependencia en el software que supondría implementar Kerberos y se da pie a la extensión con futuros mecanismos ya que su funcionamiento es independiente del protocolo empleado para la autenticación.

Junto a la implementación de este mecanismo como un *plugin* que extiende al módulo Keystone para la gestión de identidades, se ha realizado la adaptación del cliente para el servicio Swift de almacenamiento de ficheros en red. Su finalidad era poder realizar una demostración práctica de la viabilidad del mecanismo en un escenario real. Con dicha prueba ha quedado demostrado que el modelo presentado para el mecanismo es completamente funcional. Queda por tanto como una posible vía de trabajo futura la adaptación al mecanismo de los restantes clientes del entorno OpenStack.

A continuación, se ha elaborado un estudio teórico de la capacidad de usar este mecanismo para una autenticación en un entorno Kerberos Cross-Realm, permitiendo así la autenticación de usuarios federados. Las conclusión de este análisis es que dicho uso es viable sin necesidad de tener que modificar el mecanismo actual - salvo para adaptarlo al mapeo de usuarios federados a usuarios locales del entorno OpenStack de forma que se adecúe a las necesidades de uso -, gracias a que la gestión del mecanismos Kerberos Cross-Realm es realizada por la GSS-API de forma ajena al mecanismo y sin requerir pasos adicionales en la negociación entre cliente y servidor. Se deja como trabajo futuro el despliegue de un escenario con múltiples realms Kerberos para comprobar de manera prácticas nuestras afirmaciones, o por contra, encontrar alguna diferencia entre lo establecido por los modelo teórico y el resultado de la prueba.

Posteriormente, se ha realizado una prueba de concepto para la integración de Kerberos con la arquitectura ABFAB/Moonshot utilizando para ello el módulo Keystone adaptado por la Universidad de Kent. Dicha adaptación no ha sido completa ya que únicamente ha consistido en aprovechar el entorno del mecanismo Moonshot ya desplegado, para sobre él realizar la autenticación con Kerberos sin llegarse a producirse una auténtica federación en ningún caso. Se abre así otra posible vía de trabajo posterior que consistiría en la integración completa de Kerberos en el entorno ABFAB/Moonshot mediante alguna de las formas expuestas en [19].

Una vez que podemos considerar como cerrado los objetivos de estudio, diseño e implementación del trabajo, nos queda presentar nuestros resultados a la comunidad de OpenStack para que puedan ser aprovechados. Esto supondrá además un compromiso de mantenimiento y mejorar del producto desarrollado ya que es muy probable que al ser desplegado en nuevos y más variados entornos sean necesarias mejoras de implementación.

Finalmente, otra posible rama de trabajo sería probar la solución propuesta por el compañero D. Alejandro Abad Carrascosa en su Trabajo Fin de Grado 'Arquitectura Single Sign-On basada en Kerberos para el acceso a servicios federados tras autenticación en la red' para que usuarios federados puedan realizar un acceso Single Sing On en servicios kerberizados. Aprovechando su desarrollo, se abriría una nueva posibilidad en el acceso a OpenStack de forma federada y mediante la arquitectura Kerberos.

Agradecimientos

Agradecer a D. Alejandro Pérez Méndez, estudiante de tesis en la Universidad de Murcia, su gran predisposición para colaborar. Destaco en especial su ayuda a la hora de revisar los problemas de configuración en el entorno Kerberos y sus comentarios sobre las erratas presentes en [77].

He de resaltar también la tesón de los tutores de este Trabajo Fin de Grado, D. Gabriel López Millán y D. Rafael Marín López, cuyos constantes consejos y correcciones han permitido que este trabajo haya llegado a ser lo que es.

Por último, dar las gracias a la Universidad de Kent por su trabajo realizado, ya que con el software proporcionado ha sido posible superar los problemas que presenta la poca documentación del proyecto OpenStack.

Referencias

[1] WhatsApp Inc. Whatsapp: Simple. personal. mensajería en tiempo real. https://http://www.whatsapp.com/?l=es. Última consulta: 17 de Julio del 2014.

[2] Spotify Spain S.L. Spotify: Música para todos. https://www.spotify.com/es/. Última consulta: 17 de Julio del 2014.

[3] Google. Google docs: Todos tus archivos estés donde estés. https://docs.google.com/?hl=es. Última consulta: 17 de Julio del 2014.

[4] T. Mell, P. ; Grance. The nist definition of cloud computing. NIST Special Publication 800-145, National Institute of Standards and Technology, septiembre 2011.

[5] Dropbox. Dropbox: Tus archivos, estés donde estés. https://www.dropbox.com/home. Última consulta: 17 de Julio del 2014.

[6] Google. Gmail: La sencillez y facilidad de gmail en todo tipo de dispositivos. https://www.gmail.com/intl/es/mail/help/about.html. Última consulta: 17 de Julio del 2014.

[7] Inc. o sus empresas afiliadas Amazon Web Services. Amazon web services. http://aws.amazon.com/es/. Última consulta: 17 de Julio del 2014.

[8] Daniele Catteddu. *Cloud Computing: benefits, risks and recommendations for information security*. Springer, 2010.

[9] OpenStack Community. Openstack: Open source software for building private and public clouds. https://www.openstack.org/. Última consulta: 17 de Julio del 2014.

[10] Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, y otros. Oauth core 1.0 revision a. http://oauth.net/core/1.0a/. Última consulta: 16 de Julio del 2014.

[11] MIT Kerberos Team. The network authentication protocol. http://web.mit.edu/kerberos/www. Última consulta: 17 de Julio del 2014.

[12] CERN. Cern, accelerating science. http://home.web.cern.ch/. Última consulta: 16 de Julio del 2014.

[13] Tim Bell. Managing identities in the cloud. http://openstack-in-production.blogspot.com.es/2013/08/managing-identities-in-cloud.html. Última consulta: 17 de Julio del 2014.

[14] Tatu Ylonen y Chris Lonvick. The secure shell (ssh) protocol architecture. 2006.

[15] AK Bhushan. File transfer protocol. 1972.

[16] Jon Postel. Simple mail transfer protocol. *Information Sciences*, 1982.

[17] Sandro Angius. Kerberos cross-realm authentication. `http://www.dia.uniroma3.it/~afscon09/docs/angius.pdf`. Última consulta: 17 de Julio del 2014.

[18] TERENA GÉANT. What is eduroam? `https://www.eduroam.org/`. Última consulta: 17 de Julio del 2014.

[19] A. Pérez-Mendez, F. Pereniguez-García, R. Marín-López, G. López, y J. Howlett. Identity federations beyond the web: A survey. IEEE Communications Surveys and Tutorials.

[20] OASIS. Oasis security services (saml) tc. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`. Última consulta: 17 de Julio del 2014.

[21] IETF OAuth WG. Oauth 2.0. `http://oauth.net/2/`. Última consulta: 17 de Julio del 2014.

[22] David Recordon y Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16. ACM, 2006.

[23] Carl Rigney, S Willens, A Rubens, y W Simpson. Rfc2865. *Remote Authentication Dial In User Service (RADIUS), IETF*, 2000.

[24] C Rigney. Rfc2866. *RADIUS accounting*, 2000.

[25] Bernard Aboba, Larry Blunk, John Vollbrecht, James Carlson, Henrik Levkowetz, y otros. Extensible authentication protocol (eap). Technical report, RFC 3748, June, 2004.

[26] Josh Howlett, Sam Hartmann, Hannes Tschofenig, y Eliot Lear. Application bridging for federated access beyond web (abfab) architecture, 2010.

[27] Janet. Janet, about us. `https://www.ja.net/about-janet/about-us`. Última consulta: 16 de Julio del 2014.

[28] J. Linn. Generic security service application program interface. Request for Comments 1508, Internet Engineering Task Force, enero 2000.

[29] Eric Rescorla. Http over tls. 2000.

[30] RL Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, y Ken Klingenstein. Federated security: The shibboleth approach. *Educause Quarterly*, 27(4):12–17, 2004.

[31] IETF. The internet engineering task force (ietf). `http://www.ietf.org/`. Última consulta: 19 de Julio del 2014.

[32] Janet. Moonshot. `https://www.ja.net/products-services/janet-futures/moonshot`. Última consulta: 16 de Julio del 2014.

[33] Massachusetts Institute of Technology. Kerberos: The network authentication protocol. `http://web.mit.edu/kerberos/`. Última consulta: 17 de Julio del 2014.

[34] Massachusetts Institute of Technology. Massachusetts institute of technology. `http://web.mit.edu/`. Última consulta: 17 de Julio del 2014.

[35] Massachusetts Institute of Technology. Athena computing environment. `https://ist.mit.edu/athena`. Última consulta: 17 de Julio del 2014.

[36] W. Stallings. *Cryptography and Networking Security*. Prentice Hall, quinta edition, 2010.

[37] John Linn. The kerberos version 5 gss-api mechanism. 1996.

[38] Derrell Piper y Brian Swander. A gss-api authentication method for ike. *Network Working Group Internet Draft*, 2001.

[39] J. S. Hartman, Ed. ; Howlett. A gss-api mechanism for the extensible authentication protocol. Request for Comments 7055, Internet Engineering Task Force, diciembre 2013.

[40] Wikipedia. Computación en la nube. `http://es.wikipedia.org/wiki/Computaci%C3%B3n_en_la_nube`. Última consulta: 17 de Julio del 2014.

[41] Agencia NIST. National institute of standards and technology. `http://www.nist.gov/`. Última consulta: 17 de Julio del 2014.

[42] Google. Google app engine. `https://appengine.google.com`. Última consulta: 17 de Julio del 2014.

[43] G. van Rossum. *Python Language Reference Manual*. Network Theory Limited, primera edition, 2003.

[44] R. Lerdorf, K. Tatroe, y P. MacIntyre. *Programming PHP*. O'Reilly Media, Inc., segunda edition, 2006.

[45] G. Horstmann, Cay S. ; Cornell. *Core Java: Volume I - Fundamentals*. Pearson, octava edition, 2011.

[46] Google. The go programming language. `http://golang.org/`. Última consulta: 17 de Julio del 2014.

[47] MySQL AB. *MySQL Administrator's Guide*. MySQL Press, primera edition, 2004.

[48] P.A Bernstein. Adapting microsoft sql server for cloud computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, febrero 1988.

[49] Heroku. Heroku: Build, run, and scale apps. `https://www.heroku.com/`. Última consulta: 17 de Julio del 2014.

[50] Yukio Matsumoto y K Ishituka. Ruby programming language, 2002.

[51] Stefan Tilkov y Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 2010.

[52] Amazon. Amazon web services. `http://aws.amazon.com/es/`. Última consulta: 17 de Julio del 2014.

[53] Rackspace. Rackspace: the open cloud company. `http://www.rackspace.com`. Última consulta: 17 de Julio del 2014.

[54] NASA. National aeronautics and space administration agency. `http://www.nasa.gov/`. Última consulta: 17 de Julio del 2014.

[55] Inc Qumranet. Kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`. Última consulta: 17 de Julio del 2014.

[56] Citrix. Xenserver: Open source virtualization. `http://www.xenserver.org/`. Última consulta: 17 de Julio del 2014.

[57] Microsofot. Microsoft: Server and cloud platform. `microsoft.com/hyper-v`. Última consulta: 17 de Julio del 2014.

[58] Openstack Foundation. Devstack: A documented shell script to build complete openstack development environments. `http://devstack.org/`. Última consulta: 17 de Julio del 2014.

[59] Openstack Foundation. The openstack foundation. http://www.openstack.org/foundation/. Última consulta: 17 de Julio del 2014.

[60] OpenStack Community. Openstack api complete reference. http://api.openstack.org/api-ref.html. Última consulta: 17 de Julio del 2014.

[61] K. ; Ed. Zeilenga. Lightweight directory access protocol (ldap): Technical specification road map. Request for Comments 4510, Internet Engineering Task Force, junio 2006.

[62] Ed. D. Hardt. The oauth 2.0 authorization framework. Standards track, Internet Engineering Task Force, octubre 2012.

[63] P Calhoun, J Loughney, E Guttman, G Zorn, y J Arkko. Rfc 3588-diameter base protocol. *Network Working Group*, page 48, 2003.

[64] J Howlett y S Hartman. A radius attribute, binding, profiles, name identifier format, and confirmation methods for saml. 2014.

[65] Larry Zhu, Sam Hartman, y Kenneth Raeburn. Kerberos principal name canonicalization and cross-realm referrals. 2012.

[66] Shoichi Sakane, Masahiro Ishiyama, y Saber Zrelli. Problem statement on the cross-realm operation of kerberos. 2010.

[67] Openstack identity api v3: What's new in version 3.0.

[68] Openstack identity api v3: Os-oauth1 extension.

[69] Add new v3 resource to provide for kerberos authentication.

[70] Paul V Mockapetris. Domain names-concepts and facilities. 1987.

[71] RJ Systems. Dns discovery for mit kerberos v. http://www.rjsystems.nl/en/2100-dns-discovery-kerberos.php. Última consulta: 16 de Julio del 2014.

[72] Openstack Foundation. Authentication plugins. http://docs.openstack.org/developer/keystone/configuration.html#authentication-plugins. Última consulta: 17 de Julio del 2014.

[73] D. Eastlake 3rd. Rfc 4122 - a universally unique identifier (uuid) urn namespace. *Network Working Group*, 2005.

[74] BIND9.NET / BIND9.ORG. Dns, bind, dhcp, ldap and directory services. http://www.bind9.org/. Última consulta: 16 de Julio del 2014.

[75] KWSS. Identity service for openstack. https://github.com/kwss/keystone/. Última consulta: 20 de Julio del 2014.

[76] KWSS. Swift client for openstack. https://github.com/kwss/python-swiftclient. Última consulta: 20 de Julio del 2014.

[77] K. Siu. Installing the keystone server with the federated plugin. http://sec.cs.kent.ac.uk/CLASSe/server_install_guide_1.2.pdf, enero 2014. Última consulta: 17 de Julio del 2014.

[78] K. Siu. Installing the moonshot enabled swift client. http://sec.cs.kent.ac.uk/CLASSe/client.pdf, enero 2014. Última consulta: 17 de Julio del 2014.

[79] V. Giersch. Pymoonshot: A fork of pykerberos for an implementation of janet moonshot. https://github.com/gierschv/pymoonshot, 2013. Última consulta: 17 de Julio del 2014.

[80] Wireshark Foundation. Wireshark. http://www.wireshark.org/. Última consulta: 17 de Julio del 2014.

Índice de figuras

Anexos

Configuración del servidor de nombres de dominio

Fichero /etc/bind/named.conf.local:

```
zone "openstack.um" {
  type master;
  file "/etc/bind/db.openstack.um.zone";
};


zone "0.168.192.in-addr.arpa" {
  type master;
  file "/etc/bind/db.0.168.192.zone";
};
```

Fichero /etc/bind/named.conf.options:

```
options {
  directory "/var/cache/bind";
  allow-recursion {192.168.0.0/24;};
  recursion yes;
};
```

Fichero /etc/bind/db.openstack.um.zone:

```
$ORIGIN openstack.um.
$TTL 86400

@          IN   SOA ns    postmaster ( 1 6H 1H 2W 3H )
           IN   NS  ns

ns         IN   A 192.168.0.1
kdc        IN   A 192.168.0.1
kadmin     IN   A 192.168.0.1
radius     IN   A 192.168.0.1



keystone       IN   A 192.168.0.2
swift          IN A 192.168.0.2
```

Fichero /etc/bind/db.0.168.192.zone:

```
$ORIGIN 0.168.192.in-addr.arpa.
$TTL 8640
@    IN   SOA ns.openstack.um.  postmaster ( 1 6H 1H 2W 3H )

     IN   NS  ns.openstack.um.
2    IN   PTR keystone.openstack.um.
1    IN   PTR kdc.openstack.um.
```

Guión de instalación de Kerberos

```
#Instalamos los paquetes del servidor kerberos:

sudo apt-get install krb5-kdc krb5-admin-server

#Las máquinas virtuales tienen problemas llenando el pool de entropía
   de linux, para acelerar el proceso ejecutamos el siguiente comando,
   aunque esto no se debe hacer en un despliegue en producción de un
   servicio kerberos ya  que debilita la fortaleza de las claves
   utilizadas.

sudo cat /dev/sda1 >> /dev/random  &

#Mientras se ejecuta el proceso anterior creamos un realm para nuestro
   entorno. Por defecto se creará con el nombre de dominio utilizado
   para nuestra máquina, es decir, OPENSTACK.UM.

sudo krb5_newrealm

#Sustituimos el contenido del fichero /etc/krb5.conf por:

[libdefaults]
        default_realm = OPENSTACK.UM

# The following krb5.conf variables are only for MIT Kerberos.
        krb4_config = /etc/krb.conf
        krb4_realms = /etc/krb.realms
        kdc_timesync = 1
        ccache_type = 4
        forwardable = true
        proxiable = true

# The following libdefaults parameters are only for Heimdal Kerberos.
        v4_instance_resolve = false
        v4_name_convert = {
                host = {
                        rcmd = host
                        ftp = ftp
                }
                plain = {
                        something = something-else
                }
        }
        fcc-mit-ticketflags = true
```

```
[realms]
        OPENSTACK.UM = {
                kdc = kdc.openstack.um
                admin_server = kadmin.openstack.um
        }

[domain_realm]
        .openstack.um = OPENSTACK.UM
        openstack.um = OPENSTACK.UM


[login]
  krb4_convert = true
  krb4_get_tickets = false

# Entramos en el entorno de configuración de kerberos

sudo kadmin.local

# Añadimos al usuario test

add_principal -pw moonshot test

# Añadimos el servicio keystone

add_principal -randkey keystone/keystone.openstack.um

# Expertamos las claves para el servicio keystone a un fichero keytab

ktadd keystone/keystone.openstack.um

# Por último copiamos el fichero /etc/krb5.keytab en la máquina
   Keystone.
```

Guión de instalación de FreeRADIUS

```
#Dependencias: 1. openssl
sudo apt-get install openssl* -y

# 1. Descargamos las fuentes de la versión 2.2.3
wget ftp://ftp.freeradius.org/pub/freeradius/freeradius-server-2.2.3.
    tar.gz

# 2. Descomprimimos el fichero "freeradius-server-2.2.3.tar.gz"
 tar xzvf freeradius-server-2.2.3.tar.gz

# 3. Entramos al directorio creado
 cd freeradius-server-2.2.3

# 4. Modificamos el fichero "configure". Hay que sustituir todas las
    apariciones de "-lpthread" por "-pthread".

# 5. Instalamos, indicando como directorio base de instalación "/usr/
    local/freeradius" y directorio para los archivos de configuración "/
    etc"
 ./configure --prefix=/usr/local --sysconfdir=/etc
 make
 sudo make install

# 6. Damos permisos de lectura a todos en el directorio "/etc/raddb",
    de lo contrario, deberemos ejecutar como root todas las herramientas
     de cliente radius. (Este paso es para el desarrollo del trabajo
    actual, obviamente, hacer esto en un sistema en producción es un
    riesgo grave)
 sudo chmod a+x /etc/raddb/certs/ /etc/raddb/modules/ /etc/raddb/sites-
    available/ /etc/raddb/sites-enabled/
 sudo chmod a+r /etc/raddb -R


# 7. Creamos los enlaces simbólicos de los ejecutables para que estos
    se encuentren en el $PATH.
 ln -s /usr/local/freeradius/bin/* /bin
 ln -s /usr/local/freeradius/sbin/* /sbin

#8. Añadimos un usuario en \texttt{/etc/raddb/users}:
 test    Cleartext-Password := "moonshot"
```

```
#9. En el fichero /etc/raddb/sites-enabled/default, al final de la
    sección post-auth insertamos:

update reply {
  SAML-AAA-Assertion = '<saml:Assertion xmlns:saml="urn:oasis:names:tc:
   SAML:2.0:assertion" IssueInstant="2011-03-19T08:30:00Z" ID="foo"
   Version="2.0">'
  SAML-AAA-Assertion += '<saml:Issuer>urn:mace:incommon:osu.edu</saml:
   Issuer>'
  SAML-AAA-Assertion += '<saml:Subject><saml:NameID NameQualifier="max.
   feide.no" SPNameQualifier="urn:mace:feide.no" Format="urn:oasis:
   names:tc:SAML:2.0:nameid-format:persistent">UB/
   WJAaKAPrSHbqlbcKWu7JktcKY</saml:NameID></saml:Subject>'
  SAML-AAA-Assertion += '<saml:Conditions NotBefore="2007-12-10T11
   :29:48Z" NotOnOrAfter="2017-12-10T19:39:48Z"></saml:Conditions>'
  SAML-AAA-Assertion += '<saml:AttributeStatement>'
  SAML-AAA-Assertion += '<saml:Attribute NameFormat="urn:oasis:names:tc
   :SAML:2.0:attrname-format:uri" Name="portal_id"><saml:AttributeValue
   >test</saml:AttributeValue></saml:Attribute>'
  SAML-AAA-Assertion += '<saml:Attribute NameFormat="urn:oasis:names:tc
   :SAML:2.0:attrname-format:uri" Name="urn:oid
   :1.3.6.1.4.1.5923.1.1.1.7"><saml:AttributeValue>moonshot</saml:
   AttributeValue></saml:Attribute>'
  SAML-AAA-Assertion += '</saml:AttributeStatement>'
  SAML-AAA-Assertion += '</saml:Assertion>'
}

# 10. Para arrancarlo en modo debug, ejecutaremos
  /sbin/ldconfig -v
  radiusd -X
```

Código fuente de KerberosBasic

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from keystone import auth
from keystone import exception
from keystone import identity
from keystone.openstack.common import log as logging
from keystone.common import config
from datetime import date, datetime, timedelta
from lxml.etree import parse, tostring, fromstring, ElementTree
import time
import uuid
import pymoonshot
import logging

LOG = logging.getLogger(__name__)
CONF = config.CONF

class KerberosException(Exception):
    pass

class Kerberos(auth.AuthMethodHandler):
 contexts = {}
 service = CONF.auth.get('KerberosServiceHost')
 oid = '{1 2 840 113554 1 2 2}'
 identity_api = identity.Manager()

 def __exit__(self, type, value, traceback):
     for cid in Kerberos.contexts.keys():
         logging.debug('Clean GSSAPI context: %s', cid)
         self.destroyClientContext(cid)

 def setClientContext(self, cid, contextGSSAPI):
     timeout = 10
     contextGSSAPI['expires'] = datetime.now() + timedelta(seconds=
   timeout)
     Kerberos.contexts[cid] = contextGSSAPI

 def getClientContext(self, cid):
     self.cleanExpiredContextes()
     if cid in Kerberos.contexts:
         return Kerberos.contexts[cid]
     return None
```

```
43  def cleanExpiredContextes(self):
        for cid in Kerberos.contexts.keys():
45          if Kerberos.contexts[cid]['expires'] < datetime.now():
                logging.debug('Clean expired GSSAPI context: %s', cid)
47              self.destroyClientContext(cid)

49  def destroyClientContext(self, cid=None, contextGSSAPI=None, clean=
      True):
        try:
51          if cid is not None:
                if cid in Kerberos.contexts:
53                  pymoonshot.authGSSServerClean(
                        Kerberos.contexts.pop(cid)['context']
55                  )
            if contextGSSAPI is not None:
57              pymoonshot.authGSSServerClean(contextGSSAPI)
            logging.debug(
59           'Remaining contextes: %r' % Kerberos.contexts
            )
61      except Exception, err:
            logging.error(
63           'GSS clean error: %s' % err
            )
65

    def authenticate(self, context, auth_payload, user_context):
67   # Recogemos el CID
     if 'cid' in auth_payload  and auth_payload['cid'] is not None:
69    cid = uuid.UUID(auth_payload['cid']).hex
     # Si no existe, puede ser que se trate del mensaje inicial
71   elif 'phase' in auth_payload and auth_payload.get('phase') == '
      discovery':
       # Establecemos un ID de sesion
73    return {'cid':uuid.uuid4().hex, 'negotiation':'', 'service' :
      Kerberos.service}
     # En otro caso, hay un error de formato en el mensaje
75   else:
      raise exception.ValidationError(
77     attribute='cid',
       target=auth_payload
79    )

81   # Comprobamos que exista el atributo phase
     if 'phase' in auth_payload:
83    phase = auth_payload.get('phase')
```

```
85    else:
       raise exception.ValidationError(
87      attribute='phase',
        target=auth_payload
89     )

91    if phase == 'negotiate':

93     # Fase Negociacion: autenticación GSS-API Kerberos
       # Si no existe el campo negotiation hay un error de formato
95
       if not 'negotiation' in auth_payload:
97      raise exception.ValidationError(
         attribute='negotiation', target=auth_payload
99      )

101    #Cogemos el token gss-api
       negotiation = auth_payload.get('negotiation')
103    # En otro caso recogemos el contexto GSS-API asociado
       contextGSSAPI = self.getClientContext(cid)
105    # Construimos un mensaje de repuesta provisional
       resp = {'cid': cid, 'negotiation': None}
107
       try:
109     # Si no hay un contexto significa que es la primera llamada a la
      GSS-API
        if contextGSSAPI is None:
111      #Inicializamos el contexto
         contextGSSAPI = {}
113      # Primera invocación a la GSS-API que nos construye un contexto
         contextGSSAPI['state'], contextGSSAPI['context'] = pymoonshot.
      authGSSServerInit(Kerberos.service, Kerberos.oid)
115      # Comprobamos si ha habido algún error mirando el estado
         if contextGSSAPI['state'] < 0:
117       raise KerberosException(
          'authGSSServerInit returned %d: '  %state
119      )

121     # Aquí ya tenemos un contexto GSS-API construido
        # Le pasamos el token GSS-API recibido para que lo procese
123     contextGSSAPI['state'] = pymoonshot.authGSSServerStep(
         contextGSSAPI['context'], negotiation
125     )

127     # Almacenamos el contexto GSS-API para una consulta posterior
```

```
    self.setClientContext(cid, contextGSSAPI)
    resp['negotiation'] = pymoonshot.authGSSServerResponse(
     contextGSSAPI['context']
    )

    # Devolvemos el mensaje construido con el token GSS-API y el UUID
    return resp

     # Capturamos posibles excepciones lanzadas por el método
   except (Exception), err:
       LOG.exception(err)
       raise exception.Unauthorized('Unauthorized user')

  # Si el campo phase es complete se ha terminado la negociacion
  elif  phase == 'complete':

   # Comprobamos que el contexto corresponde a una negociacion
   contextGSSAPI = self.getClientContext(cid)
   if contextGSSAPI is None:
    raise exception.Unauthorized(
     'Not a valid context for this cid'
    )
   # Buscamos el ID del usuario asociado a la negociacion
   userName =  pymoonshot.authGSSServerUserName(
    contextGSSAPI['context']
   )

   # Mapeo para usuarios autenticados con Kerberos Cross-Realm
   if not userName.endswith('@OPENSTACK.UM'):
    userName = 'federado'

   usuario = Kerberos.identity_api.get_user_by_name(userName,'default')
   user_context['user_id'] = usuario['id']
   return None

  else:
   raise exception.ValidationError(attribute='phase',target=
   auth_payload)
```

Código de KerberosBasic en el cliente Swift

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import copy
import json
import urllib3
import logging
import httplib
import platform
import pymoonshot
import sys

from urlparse import urlparse, urlunparse, urljoin


LOG = logging.getLogger('swiftclient')
LOG.addHandler(logging.StreamHandler())
LOG.setLevel(logging.INFO)

class KerberosException(Exception):
    pass

class KerberosNegotiation(object):
    def __init__(self, keystoneEndpoint, requestPool, tenant_name):
        self.tenant_name = tenant_name
        self.context = None
        self.serviceName = ''
        self.mechanism = '{1 2 840 113554 1 2 2}'
        self.step = 0
        self.requestPool = requestPool
        self.keystoneEndpoint = keystoneEndpoint
        self.idpResponse = {}

    def negotiation(self):
        ## Envío del mensaje KerberosBasic (Discovery Request)
        body = '{"auth": {"identity": {"methods": ["kerberosBasic"], "
    kerberosBasic" : {"phase":"discovery"}}}}'
        headers = {"Content-type": "application/json"}
        resp  = json.loads(self.requestPool.urlopen('POST', self.
    keystoneEndpoint, body = body, headers = headers).data)

        ## Recepción del mensaje KerberosBasic(Discovery Response)
      discoveryResponse = resp['error']['identity']['kerberosBasic']
```

```
42      self.idpResponse = {'cid':discoveryResponse['cid'],'negotiation':'
    '}

44      ## Establecimiento del contexto gss-api
        result, self.context = pymoonshot.authGSSClientInit(
    discoveryResponse['service'],
46              pymoonshot.GSS_C_MUTUAL_FLAG | pymoonshot.
    GSS_C_SEQUENCE_FLAG, self.mechanism)

48          if result != 1:
                raise KerberosException('authGSSServerInit returned result
    %d' % result)
50          negotiation = pymoonshot.AUTH_GSS_CONTINUE

52          # Envío de los mensajes KerberosBasic(negotiate)
            while negotiation != pymoonshot.AUTH_GSS_COMPLETE:
54              negotiation = self.negotiationStep()


56
            ## Fin de la negociacion GSS-API
58          LOG.info("Authentication successful using \"%s\" kerberos
    identity",
                pymoonshot.authGSSClientUserName(self.context))
60
      ## Mensaje final para recibir el token como respuesta
62    headers = {'Content-Type':'application/json'}
        body = json.dumps({'auth':{'tenantName':self.tenant_name,
64          'identity':{'methods':['kerberosBasic'],
                        'kerberosBasic':{'phase':'complete',
66                                         'negotiation': None,
                                           'cid': self.idpResponse.
    get('cid',None)}}}})
68
            resp = self.requestPool.urlopen('POST', self.keystoneEndpoint,
70                          body = body, headers = headers)

72    self.idpResponse =  json.loads(resp.data)

74    url = None
      ## Devolvemos el token id y la url del servicio
76    if "token" in self.idpResponse and "catalog" in self.idpResponse["
    token"]:
       for cat in self.idpResponse["token"]["catalog"]:
78      if cat["type"] == "object-store":
          for endp in cat["endpoints"]:
```

```
80          if endp["interface"] == "public":
             url = endp["url"]
82      if url == None:
         url = self.keystoneEndpoint
84
        return url, resp.getheader("x-subject-token")
86


88      def negotiationStep(self):
            result = pymoonshot.authGSSClientStep(self.context, self.
    idpResponse['negotiation'])
90      # Se le pasa a la GSS-API y se obtiene el GSS-TOKEN
            idpNegotiation = pymoonshot.authGSSClientResponse(self.context)
    ;
92          ## Que idpNegotation sea no vacio significa que la negociacion
    no ha terminado.
            if idpNegotiation is not None:
94          ## Enviamos al servidor un mensaje con el challenged en el campo
    negotiation.
                server_resp = self.negotiationRequest(idpNegotiation,
    self.idpResponse.get("cid", None));
96              # Obtenemos la respuesta
                self.idpResponse = {"negotiation": server_resp["error"
    ]["identity"]["kerberosBasic"]["negotiation"],
98               "cid": server_resp["error"]["identity"]["kerberosBasic
    "].get("cid", None)}
            return result
100

        def negotiationRequest(self, body, cid = None):
102         headers = {'Content-Type':'application/json'}
            body = json.dumps({'auth':{
104             'identity':{
                    'methods':['kerberosBasic'],
106                 'kerberosBasic':{
                        'phase':'negotiate',
108                     'negotiation':body,
                         'cid': cid}}}})
110         return json.loads(self.requestPool.urlopen('POST', self.
    keystoneEndpoint, body = body, headers = headers).data)
```