UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA

TRABAJO FIN DE GRADO

Implementación de un mecanismo de fragmentación para atributos grandes en RADIUS

Junio 2014

Autor: David Cuenca García
Directores: Rafael Marín López y Gabriel López Millán

Gracias a Rafael Marín y Gabriel López por darme la oportunidad
de realizar este trabajo y confiar en mi para llevarlo a cabo,
así como por su disponibilidad y ayuda durante todo el proceso.
Gracias a Alejandro Pérez por su gran colaboración
y ayuda en la fase de implementación.

Gracias a María Noelia, por estar ahí cada día, por prestarme su apoyo
y ayudarme en las últimas fases de este trabajo.

Índice de Contenidos

Resumen 7

Extended Abstract 9

1. Introducción 13

2. Estado del arte 16
2.1 AAA 16
2.2 RADIUS 17
2.2.1 Autenticación y autorización 17
2.2.2 Estructura del paquete 20
2.2.3 Seguridad 21
2.2.4 RADIUS y EAP 22
2.3 EAP 22
2.4 SAML 24
2.5 ABFAB/Moonshot 28

3. Análisis de objetivos y metodologías 32

4. Diseño e implementación de la solución propuesta 33
4.1 Motivación del diseño 33
4.2 Descripción del proceso de fragmentación 34
4.2.1 Concepto de chunk y atributos de fragmentación 34
4.2.2 Fases de la fragmentación 35
4.2.2.1 Flujo de pre-autorización 37
4.2.2.2 Flujo de post-autorización 39
4.2.3 Discusión de los aspectos de diseño 41
4.3 Implementación de la solución 42
4.3.1 Software utilizado 42
4.3.1.1 FreeRADIUS 42
4.3.2 Implementación de servidor 43
4.3.2.1 Estructuras de datos para la fragmentación 43
4.3.2.2 Funciones auxiliares 44
4.3.2.3 Funciones principales 44
4.3.2.3.1 Pre-autorización 44
4.3.2.3.2 Post-autorización 46
4.3.2.4 Integración con FreeRADIUS 48
4.3.2.5 Gestión de datos de pre-autorización y post-autorización 49
4.3.3 Implementación en el cliente 51
4.3.3.1 Funciones y estructuras de datos para la fragmentación 51
4.3.3.1.1 Pre-autorización 52
4.3.3.1.2 Post-autorización 52
4.3.3.2 Radclient y radeapclient 53
4.3.3.3 ABFAB/Moonshot 54

5. Validación de la implementación 58
5.1 Flujo de pre-autorización 59
5.2 Flujo de post-autorización 61

6. Conclusión y vías futuras 63

7. Referencias 64

Índice de Figuras

Figura 1: Intercambio básico de mensajes RADIUS. 19
Figura 2: Escenario básico de autenticación con RADIUS. 19
Figura 3: Proceso de roaming RADIUS. 20
Figura 4: Estructura atributo RADIUS. 21
Figura 5: Flujo de autenticación EAP mediante RADIUS. 24
Figura 6: Escenario Web-SSO con SAML. 26
Figura 7: Estructura mensajes SAML. 27
Figura 8: Ejemplo de XML Signature 28
Figura 9: Flujo de autenticación en ABFAB/Moonshot con SAML 30
Figura 10: Flujo de mensaje RADIUS con fragmentación 36
Figura 11: Contenido del paquete Access Request original 37
Figura 12: Flujo de mensajes en pre-autorización 38
Figura 13: Contenido del paquete Access Accept original 39
Figura 14: Flujo de mensajes en post-autorización 40
Figura 15: Escenario para la validación de la implementación 58
Figura 16: Mensajes RADIUS en el flujo de pre-autorización con Moonshot 59
Figura 17: Contenido del chunk de pre-autorización 60
Figura 18: Contenido de la respuesta al chunk de pre-autorización 60
Figura 19: Contenido del último chunk de pre-autorización 60
Figura 20: Mensajes RADIUS en el flujo de post-autorización con Moonshot 61
Figura 21: Contenido del primer chunk de post-autorización 61
Figura 22: Contenido de la respuesta al chunk de post-autorización 62

Resumen

En la actualidad, hay muchos servicios y recursos en Internet, los cuales hacen uso de una gran cantidad de datos. Todo esto provoca que exista la preocupación y necesidad de crear procesos de autenticación seguros y fáciles para el usuario.

Uno de los escenarios más utilizados para la gestión del acceso a las redes o servicios se basa en un protocolo AAA (Authentication, Authorization y Accounting). Este tipo de protocolo permite controlar el acceso y el uso de los servicios de un dominio, validando las credenciales presentadas por el usuario y comprobando si dispone de autorización para utilizar el recurso solicitado.

Actualmente se está integrando la infraestructura AAA para controlar el acceso a servicios de aplicación en algunos escenarios. Uno de los protocolos más utilizados es RADIUS, ya que permite la gestión del acceso a los servicios y recursos. Por ejemplo, es muy utilizado en la red *eduroam* para gestionar el acceso a la red y a los recursos. En estos escenarios también se requiere enviar información adicional para la autenticación y autorización a la hora de acceder a los servicios (como Telnet, FTP, etc).

Puede ser necesario enviar esta información antes de la autenticación, ya que el cliente tiene la necesidad de enviar datos de autorización hacia el servidor (*pre-autorización*); o también puede ser necesario que el servidor quiera enviar información hacia el cliente después de la autenticación (*post-autorización*). Esta información adicional puede ser enviada en diferentes formatos como JSON, SAML, etc. Y puede tener gran tamaño debido a la longitud de los atributos, o por ir cifrado o firmado digitalmente.

Por lo tanto, es necesario el envío de datos de gran tamaño dentro de paquetes RADIUS para gestionar el acceso a los servicios de aplicación. Sin embargo, este protocolo tiene una limitación en el tamaño máximo de paquete, no pudiendo exceder de 4096 bytes. Debido a esta limitación y a la integración de RADIUS para gestionar el acceso a servicios de aplicación, en los casos en los que el paquete RADIUS exceda de 4096 bytes será descartado, haciendo que no sea posible el envío de esta información.

Por lo tanto, la solución puede consistir en fragmentar ese paquete en otros más pequeños que no excedan el límite de tamaño. Dada la existencia de algunas soluciones que no solucionan la problemática, dentro del grupo de trabajo RADEXT del IETF se comenzó a trabajar en la creación de un mecanismo que permite la fragmentación de paquetes RADIUS con un tamaño mayor que 4.096 bytes. Esta solución no requiere modificar elementos intermedios como proxies y es interoperable con infraestructuras RADIUS existentes. Sólo es necesario modificar los elementos que desean enviar o recibir paquetes fragmentados. Más concretamente, define un protocolo que permite el envío de paquetes RADIUS con información superior a 4096 bytes, tanto en pre-autorización como en post-autorización. Para ello define nuevos atributos RADIUS y reutiliza otros existenes para gestionar el proceso de fragmentación.

Como primer paso, este documento define una implementación del proceso de fragmentación propuesto para crear una versión funcional que demuestre su aplicabilidad y validez. En concreto, se ha implementado en FreeRADIUS, integrando la funcionalidad de pre-autorización y post-autorización. A su vez, se ha implementado

un cliente básico que da sorpote a ambos casos para validar la implementación y el diseño de la solución. Para esta implementación se ha considerado SAML como tecnología para representar la información de autorización.

Por otra parte, una herramienta que permite resolver el problema del acceso de modo federado a servicios de aplicación es ABFAB/Moonshot. Esta arquitectura permite el acceso federado a la aplicaciones no-web (servicios de aplicación) utilizando la infraestructura AAA. Un acceso federado es aquel que permite que la autenticación y la autorización de un usuario se realice en una entidad distinta a la que proporciona el servicio, ya sea en el mismo dominio o uno diferente. ABFAB/Moonshot utiliza RADIUS para llevar a cabo el proceso de autenticación entre las aplicaciones cliente y el servidor de autenticación. Por lo tanto, igual que antes, existe la necesidad de intercambiar información de autorización entre el servicio y el servidor de autenticación para validar la identidad y permisos de un usuario, no siendo posible si supera 4096 bytes. Más concretamente, ABFAB/Moonshot puede tener la necesidad de enviar grandes datos de pre-autorización y de post-autorización usando la tecnología SAML. Por lo tanto, la propuesta de fragmentación presentada anteriormente soluciona este problema de ABFAB/Moonshot.

Debido a esto, en este trabajo también se ha decidido integrar la propuesta de fragmentación en ABFAB/Moonshot para solucionar el problema dentro de un caso de uso real. Este software integra un cliente RADIUS para realizar la autenticación federada. En dicho cliente incorporaremos los cambios necesarios para permitir el envío de datos de gran tamaño.

Gracias a las pruebas realizadas tras la integración en FreeRADIUS y ABFAB/Moonshot podremos ver que la propuesta de fragmentación permite solucionar el problema planteado al inicio de este trabajo, permitiendo enviar datos de autorización de gran tamaño (superiores a 4096 bytes) dentro de paquetes RADIUS.

Extended Abstract

At the present time, there are many services and resources on the Internet. This fact, together with the big amount of sensitive data that exist on the network, requires authenticated and authorized access. Thus, there is a need for creating secure authentication and authorization process secure and easy for the user.

One of the most used sceneries for managing access to networks or services is based on a AAA (Authentication, Authorization and Accounting) protocol. This protocol allows access control and use of the services of a domain. This type of protocol supports three basic services. The first, Authentication, that verifies the user's identity. Authorization that checks the user's permissions and conditions on the resource. Finally, Accounting manages the use of the resources and services of the domain by the user.

Now, the infrastructure AAA is integrated for checking the authentication and authorization when one user wants access to a service. The most used protocol is RADIUS. This protocol permits management the access to the services and resources. For example, it is very used on the network eduroam. Now, this network incorporates the access to services of applications that are different of web as Telnet, Cloud, etc.

For example, suppose that a student's stay at the University of Murcia, and now she can use eduroam to access the network using the credentials of their home university. The current trend is that the user can use any other service in the same way when using the network, i.e. using their university credentials to access.

The access to these services is required to send some additional information for authentication and authorization. Sending this information may be necessary before to authentication, because the client may need to set requirements on the authentication process. This is called pre-authorization. But also can be sent after authentication, at which point the server may wish to send data to the client (for example, user attributes with additional information for the authorization process). This is called post-authorization. This additional information can be sent in different formats such as JSON, SAML, etc.. and it may have big size large because of the length of the attributes, or to go encrypted or be digitally signed.

Therefore, a need exists to support sending of large data within RADIUS packets in order to manage access to application services. However, this protocol has a limitation on the maximum packet size, which cannot exceed 4096 bytes according to the standard. Because of this limitation and the integration of RADIUS to manage access to application services, a problem arises when it comes to sending data which causes the RADIUS packets to be discarded if they exceed this length.

As the need exists, clearly we must find a solution for this problem. The solution is to fragment the RADIUS packet in smaller chunks which allow send a big data of authorization. There are differents approaches that allow fragment a RADIUS packet, but none gives a viable solution because it doesn’t contemplate that the packet may exceed 4096 bytes.

For this reason, within RADEXT Working Group (WG) in the IETF there is a process underway to standardize a mechanism which allows the fragmentation of RADIUS packets with a size greater than 4096 bytes. In the definition of this protocol is participating staff of the University of Murcia. This solution does not require to modify

intermediate elements as RADIUS proxies and it is interoperable with the current infrastructure. You just need to modifiy the entities which want send or receive fragmented packets. It specifically permits sending RADIUS packet with size greater than 4096 bytes with authorization data in the pre-authorization and the post-authorization. For this, the solution fragments the RADIUS packet in chunks which have a size less than 4096 bytes.

As a first step, this document defines an implementation of the design embodied in the proposed fragmentation to validate and create a functional version to demonstrate its applicability. Specifically, the proposal has been implemented in FreeRADIUS, integrating functionality of pre-authorization and post-authorization. In turn, we have implemented a basic client that processes flows described in the proposal to validate the design. For this implementation it has been considered SAML as technology to represent the information of authorization, which may cause the need for fragmentation.

The motivation of the design of fragmentation solution was because not existing previous solution permits sending RADIUS packets with a size higher than 4096 bytes. For example, a posible solution is remove the restriction of 4096 bytes as maximum packet size. But, this requires updating all elements of the RADIUS infrastructure (server and proxies), which makes it unfeasible.

For this reason, the solution of fragmentation was created and we have implemented it in this work. First, we have defined how to implement the fragmentation. To send a RADIUS packet over 4096 bytes we have to fragment the original packet in chunks. These chunks are standard RADIUS packets, and therefore, perfectly valid for any RADIUS infrastructure existing. They are responsible for sending the fragmented data. They have several characteristics. For example, they have a same type as the original packet, transport a subset of the attributes to the original packet and they are exclusives of fragmentation process. Therefore they will be marked by two new RADIUS attributes (except the last chunk) indicanting the state of fragmentation. From these chunks, we may reconstruct the original RADIUS packet whose size has exceded the limit imposed by RADIUS.

The attributes of fragmentation will be used by the chunks to signal the fragmentation process. There are two attributes that are defined in the proposal for the fragmentation process. One attribute is Frag-Status. This is used to determine the state of the fragmentation process. For example, value 2 (More-Data-Pending) indicates whether data fragmented are sent; value 3 (More-Data-Request) if they are asking for more; or if it is not pressence this attribute, it indicates that the process is finished. Thus, it should appear whenever there is an exchange of fragmentation except the last chunk sent, because it is not really a fragmented packet.

Other new attribute is Proxy-State-Len, that indicates the size that the client must reserve when sends a chunk, because some proxies in the path can add Proxy-State attribute. Other attributes which are used, but exist in the original definition of RADIUS, are State (to associate requests and responses in a process of fragmentation), Message-Authenticator (to add security to chunks) and Service-Type (to indicate that the authentication process is in progress).

For the implementation of the fragmentation process we will add a number of methods that will support the two possible cases of fragmentation. First case is when the user needs to send big data of authorization to the server (pre-authorization) before the authentication. He will check if the packet is bigger than 4096 bytes for starting the

fragmentation process. If it is necessary, the client will fragment the original packet creating a chunk, which is less than 4096 bytes. The attributes that have not been sent are stored on the client. This chunk is sent and when the server receive this chunk, it is processed, saving the attributes received and building the answer, which have the fragmentation attributes for the client to continue the process. The next step will be when the client receives the packet sent by the server. The client will try to send all attributes that it could not before. If these attributes do not exceed 4096 bytes, they will be sent and the fragmentation process will end. But if these attributes exceed the limit (4096 bytes), it is necessary to continue with the fragmentation process, and the client will repeat the same steps for sending a chunk. Similarly, the server will respond will send a packet for continuing with the fragmentation process. Therefore, to add support for pre-authorization we need to modify the source code of FreeRADIUS and the source code of the client. We have created a few functions to perform the functionality.

On the other hand, the other case it is when the server needs to send big data of authorization to the client after the authentication (post-authorization). Now, the server will build the packet with big data and will check if it is bigger than 4096 bytes for starting the fragmentation process. If it is necessary, the server will fragment the original packet creating a new chunk and the attributes that have not been sent will remain stored on the server. This chunk is sent and when the client receives this chunk, it is processed saving the attributes received and sending the answer with attributes of fragmentation for continuing the process of fragmentation with the server. As before, to add support for pre-authorization we need to modify the source code of FreeRADIUS and the source code of the client. We have created a few functions to perform the functionality.

Once you have done this, the next step will be when the server receives the response sent by the client. The server will try to send all attributes that have stored. If these attributes do not exceed 4096 bytes, they will be sent and the fragmentation process will end. But if these attributes exceed the limit it is necessary to continue with the fragmentation process, and the server will repeat the same steps for sending a chunk. Similarly, the client will respond similarly, and will send a packet to continue with the fragmentation process.

In this document, we have created an implementation that supports the two cases just explained. Therefore, with this implementation we resolve the problem that prevents sending RADIUS packets that are more than 4096 bytes from being sent. In addition we have developed a basic client for testing whether the two cases met the specification defined in the document of fragmentation for RADIUS.

On the other hand, a tool to solve the problem of accessing to federated application services, as described in the above example, is ABFAB/Moonshot. This architecture enables federated access to non-web applications (application services) by using AAA infrastructure. A federated access is one that allows authentication and authorization of a user it is performed in a separate entity that provides the service, either in the same or a different domain. More specifically, ABFAB/Moonshot uses RADIUS to carry out the authentication process between the client and server applications. Therefore, as before, there is a need to exchange authorization information between the service itself and the authentication server to validate the identity and permissions of a user of that service. The term ABFAB (*Application Bridging for Federated Access Beyond web*) refers to IETF working group that is standardizing this technology while Moonshot refers to the developing community that is pursuing the implementation of the protocols defined in ABFAB.

Specifically, ABFAB/Moonshot can have the need to send big data pre-authorization and post-authorization, using SAML technology. Therefore, the proposal presented above solves the fragmentation problem ABFAB/Moonshot.

Given the potential deployment ABFAB/Moonshot in this TFG we have integrated our implementation of RADIUS fragmentation in ABFAB/Moonshot implementation. This software integrates a RADIUS client to perform federated authentication. A RADIUS client that incorporates the requirements and the necessary changes to allow sending large data. So we can move this functionality to a fragmentation scenario in which it may be used widely.

Finally, we have tested our implementation in a realistic use case. For this, we have run a client with ABFAB/Moonshot and one server with FreeRADIUS, both with support for fragmentation. More specifically we have tested both cases in which we can send a big data of authorization. First, we have tested the pre-authorization and afterwards we have tested the post-authorization. In addition, we have tested the scenario adding a RADIUS proxy without fragmentation support. So we can test and validate the goal that indicates that the solution must be transparent to proxies.

Thanks to these tests, we can see that the implementation of the proposal of fragmentation allows showing that both fragmentation and its design are satisfactory, allowing to solve the problem of sending big data of authorization on RADIUS packets. In turn, it has been proven a viable solution given the current infrastructure RADIUS, causing minimal impact on it.

1. Introducción

En la actualidad cada vez son más los servicios y recursos que podemos encontrar en Internet. Con este gran auge y la gran cantidad de datos sensibles que podemos encontrar, cada vez se hace más necesaria la gestión de las identidades de los usuarios y la gestión del acceso a la red. En este sentido, la gran demanda de estos servicios provoca la preocupación y la necesidad de crear procesos de autenticación y autorización seguros, robustos y fáciles de utilizar para el usuario que permitan controlar el acceso a estos servicios.

Generalmente, uno de los escenarios más utilizados para el control de acceso a redes o servicios se basa en un protocolo AAA. Un protocolo AAA (Authentication, Authorization y Accounting) [52] da la posibilidad del control de acceso y el uso de los servicios dentro de un dominio, así como controlar el uso de los recursos que hace un usuario. Se despliega en forma de infraestructura, dando soporte a tres servicios: el primero es *Autenticación (Authentication)*, proceso mediante el cual se verifica la identidad de un usuario; en segundo lugar encontramos *Autorización* (*Authorization)*, que es el proceso que permite comprobar los permisos y las condiciones que tiene un usuario sobre un servicio; por último tenemos *Contabilidad* (*Accounting),* que permite determinar el uso que ha hecho el usuario de los recursos y servicios.

En la actualidad, también se está comenzando a integrar la infraestructura AAA en el acceso a servicios de aplicación. Para ello existen dos tecnologías AAA que pueden realizar esta función. Por un lado tenemos RADIUS [2], protocolo que de una manera sencilla gestiona el acceso a servicios y recursos. También existe DIAMETER [49], tecnología más actual, que implementa las mismas cualidades que RADIUS y ofrece algunas mejoras. Sin embargo, RADIUS está más extendido y actualmente es utilizado ampliamente en entornos como *eduroam* [50] para llevar a cabo el control de acceso a la red. Es por ello que se piensa en RADIUS a la hora de llevar a cabo la funcionalidad de permitir el acceso a servicios de aplicación, tales como Telnet [53], FTP [54], Cloud [30], etc.

Por ejemplo, supongamos un estudiante que está de estancia en la Universidad de Murcia, y que actualmente puede hacer uso de *eduroam* para obtener acceso a la red usando las credenciales de su universidad origen. La tendencia actual es que el usuario pueda usar cualquier otro tipo de servicio del mismo modo que usa la red, es decir, utilizando sus credenciales de su universidad origen para obtener acceso.

En el acceso a estos servicios se hace necesario el envío de cierta información adicional para la autenticación y autorización. El envío de esta información podría ser necesario antes de realizar la autenticación, debido a que el cliente tenga la necesidad de enviar datos de autorización hacia el servidor (por ejemplo, para establecer requisitos sobre el proceso de autenticación). Esto se denomina pre-autorización. Pero también puede que sea enviada después de la autenticación, momento en el cual el servidor puede estimar oportuno enviar datos hacia el cliente (por ejemplo, atributos de usuario con información adicional para el proceso de autorización). Esto se denomina post-autorización. Esta información adicional puede enviarse en diferentes formatos, como JSON [42], SAML [9], etc. y puede tener gran tamaño, debido a la longitud de los atributos, o a que vaya cifrada o firmada digitalmente.

Por lo tanto, existe la necesidad de soportar el envío de datos de gran tamaño dentro de paquetes RADIUS para gestionar el acceso a estos servicios de aplicación. Sin embargo, este protocolo tiene una limitación del tamaño máximo del paquete, no

pudiendo superar 4096 bytes, según el propio estándar. Debido a esta limitación y a la integración de RADIUS para gestionar el acceso a servicios de aplicación, se plantea un problema a la hora del envío de la información que provoca que los paquetes RADIUS con datos de gran tamaño sean descartados si superan los 4096 bytes, no siendo posible su envío. Para solucionar este problema es necesario fragmentar el paquete original en varios trozos de menor tamaño que sí pueden ser enviados. Por ejemplo, existen varios enfoques [28] que permiten la fragmentación de los paquetes RADIUS. Pero ninguno de estos es útil debido a que o no contemplan la posibilidad de fragmentar un paquete que exceda los 4096 bytes o no dan soporte a la fragmentación en un flujo de pre-autorización o post-autorización como los descritos anteriormente.

Debido a esto, dentro del grupo de trabajo RADEXT [48] del IETF [41] se está llevando a cabo la estandarización de un protocolo [31] que permita la fragmentación de paquetes RADIUS con un tamaño superior a 4096 bytes. En la definición de este protocolo está participando personal de la Universidad de Murcia.

Esta solución hace posible el envío de datos genéricos de autorización de gran tamaño por parte del cliente hacia el servidor (pre-autorización) y viceversa (post-autorización). No impone requisitos adicionales a los administradores de los escenarios RADIUS, ni la modificación de la configuración de elementos intermedios como routers, cortafuegos, etc. A su vez, esta propuesta es transparente con entidades RADIUS proxy existentes en la infraestructura que no incorporan esta solución de fragmentación, siendo únicamente necesario actualizar los elementos que necesiten enviar o recibir datos de gran tamaño.

Como primer paso, este Trabajo Fin de Grado define una implementación del diseño plasmado en la propuesta de fragmentación para validarlo y crear una versión funcional que demuestre su aplicabilidad. En concreto, se ha implementado la propuesta en un servidor FreeRADIUS [11], integrando en él la funcionalidad de los flujos de pre-autorización y de post-autorización. A su vez, se ha implementado un cliente básico que procese los flujos descritos en la propuesta que permite probar un escenario muy simplificado para validar la implementación y el diseño. Para esta implementación se ha considerado SAML como tecnología para representar la información de autorización necesaria y que puede provocar la necesidad de fragmentación.

Una herramienta que permite solucionar el problema del acceso de modo federado a servicios de aplicación, del modo descrito en el ejemplo anterior es ABFAB/Moonshot [12]. Esta arquitectura permite el acceso federado a servicios de aplicación que no son web (servicios de aplicaciones) mediante infraestructuras AAA. Un acceso federado es aquel que permite que la autenticación y autorización de un usuario se realice en una entidad diferente al que proporciona el servicio, sea en el mismo dominio o en uno diferente. Más concretamente, ABFAB/Moonshot utiliza RADIUS para llevar a cabo el proceso de autenticación entre las aplicaciones cliente y el servidor de autenticación. De este modo, al igual que antes, existe la necesidad del intercambio de información de autorización entre el propio servicio y el servidor de autenticación que permita validar la identidad y los permisos de un usuario sobre ese servicio. El término ABFAB [3] hace referencia al grupo de trabajo del IETF que está estandarizando esta tecnología, mientras que Moonshot [10] hace referencia a la comunidad de desarrolladores que está llevando a cabo la implementación de los protocolos definidos en ABFAB.

En concreto, ABFAB/Moonshot presenta la necesidad del envío de datos de pre-autorización y de post-autorización que pueden ser de gran tamaño, utilizando para

ellos tecnología SAML. Por lo tanto, la propuesta de fragmentación antes presentada resuelve el problema para ABFAB/Moonshot.

Teniendo en cuenta las posibilidades de despliegue que ofrece ABFAB/Moonshot, en este TFG también se ha llevado a cabo la integración del proceso de fragmentación en ABFAB/Moonshot. Este software integra un cliente RADIUS para llevar a cabo la autenticación federada. A dicho cliente RADIUS incorporaremos los requisitos y los cambios necesarios para permitir el envío de datos grandes de autorización. De modo que trasladamos esta funcionalidad de fragmentación a un escenario en el cual podrá ser usada ampliamente.

Como analizaremos posteriormente, la implementación de la propuesta va a permitir demostrar que tanto la solución de fragmentación como el diseño de la implementación son satisfactorios, permitiendo solucionar la problemática del envío de datos de autorización de gran tamaño en un escenario RADIUS. A su vez, se demostrará que es una solución viable teniendo en cuenta la infraestructura actual RADIUS y que provoca un mínimo impacto sobre la misma.

En la Sección 2 de este trabajo se describe el estado del arte de las tecnologías relacionadas con este trabajo. En el la Sección 3 describimos los objetivos planteados para este trabajo y la metodología que se ha seguido. En la Sección 4 veremos el diseño llevado a cabo para la implementación y los aspectos más importantes de la misma, con detalles acerca de cómo se ha trasladado todo lo descrito en la propuesta a una implementación tanto en el servidor como en el cliente. Por último en la Sección 5 podemos encontrar unas pruebas que permiten la validación de la implementación y en la Sección 6 las conclusiones de este Trabajo Fin de Grado.

2. Estado del arte

Las tecnologías que van a ser utilizadas para la realización de este trabajo son las descritas en la sección anterior. Por un lado, utilizaremos RADIUS como protocolo AAA. Por otro lado, usaremos ABFAB/Moonshot como arquitectura para el control de acceso a servicios de aplicación de modo federado. Concretamente el software que implementa ABFAB/Moonshot actuará como cliente RADIUS a la hora de realizar la autenticación, y nos permitirá probar la implementación y su correcto funcionamiento. En nuestro escenario, para autenticar a un usuario vamos a utilizar el protocolo EAP, y dentro de este proceso, vamos a necesitar intercambiar datos de autorización que nos permitan saber más acerca de los atributos del usuario a la hora de utilizar ABFAB/Moonshot. Para ello vamos a utilizar SAML.

2.1 AAA

Un *protocolo AAA* [8] se encuentra principalmente en escenarios entre el punto de acceso a la red al que se quiere conectar un usuario y el servidor que protege el acceso a la red. Una tecnología *AAA* (Authentication, Authorization y Accounting) *permite* el control de acceso y el uso de los servicios dentro de un dominio, así como controlar el uso de los recursos a los que accede un usuario. Se despliega en forma de infraestructura, dando soporte a estos tres servicios, que, desde un punto de vista básico, cada uno consiste en:

- Autenticación (Authentication). Proceso mediante el cual se verifica si un usuario tiene acceso o no a un servicio, verificando las credenciales que proporciona mediante las cuales identifica su identidad.

- Autorización (Authorization). Proceso mediante el cual se indican los permisos de un usuario para acceder a un servicio y en qué condiciones puede hacerlo.

- Contabilidad (Accounting). Proceso mediante el cual es posible determinar qué uso de los recursos ha hecho el usuario dentro de la red. Por ejemplo, cantidad de datos enviados y recibidos, tiempo de conexión a la red, etc. Es de utilidad a la hora de realizar estimaciones, gestión de la red y tarificación por el uso de la misma.

Las entidades que encontramos en una arquitectura AAA son:

- Usuario. Es la persona interesada en acceder a un servicio ofrecido por un operador de red.

- NAS. Es un elemento *hardware* que permite interconectar a un usuario con un servidor AAA que será el encargado de conceder el acceso a un determinado servicio o recurso. Generalmente este dispositivo será un equipo de acceso a la red o un router.

- Servidor AAA. Es la entidad que contiene la información para poder autenticar a los usuarios que requieren el permiso para acceder a los servicios que pone a disposición la arquitectura AAA. Esta información es utilizada por esta entidad para autenticar y autorizar al usuario y permitirle el acceso al recurso o servicio solicitado.

Extrapolando estas entidades a nuestro escenario, que presenta la problemática a resolver, tendremos a RADIUS como protocolo de transporte entre el NAS y el servidor AAA, y a *EAP como mecanismo de autenticación entre el cliente y el servidor AAA*. Esta será la forma de intercambiar la información de autorización entre un cliente y el servidor AAA.

A continuación vamos a explicar más detalladamente cada parte de la infraestructura presentada.

2.2 RADIUS

Es un protocolo AAA que permite gestión de la autenticación, autorización y accounting de forma centralizada para los usuarios que se conectan y quieren usar los servicios de la red. Está descrito en el RFC 2865 [2] para autenticación y autorización, y en el RFC 2866 [13] para accounting. Fue propuesto a principios de los años 90 para gestionar accesos remotos mediante redes telefónicas. En la actualidad es ampliamente utilizado tanto en ambientes inalámbricos como cableados para gestionar la autenticación y autorización principalmente de entornos federados. Consta de un modelo cliente/servidor utilizando UDP [14] como protocolo de transporte. Con gran aceptación y muy extendido, se implantó antes de ser estandarizado y actualmente sigue implantado en muchas arquitecturas AAA.

Las entidades que encontramos dentro de RADIUS son:

- Servidor RADIUS. Se corresponde con el servidor AAA visto en la sección 2.1.

- NAS (Network Access server). Es el punto de conexión entre el usuario que quiere acceder al servicio y la infraestructura AAA. Se corresponde con el NAS visto en la sección 2.1.

- Cliente RADIUS. Generalmente lo encontraremos en el NAS. Se encarga de enviar las peticiones de autenticación al servidor RADIUS y de conceder el acceso al usuario en función de lo recibido del servidor.

- Proxy RADIUS. Entidad que se encuentra en la infraestructura AAA y que se encarga de encaminar las solicitudes hacia los dominios correspondientes para realizar la autenticación. A menudo puede encontrarse en el mismo equipo que el servidor RADIUS.

A continuación describimos el flujo para autenticación y autorización en el protocolo RADIUS.

2.2.1 Autenticación y autorización

El usuario o equipo que quiere ser autenticado para el acceso a una red envía una solicitud de acceso para, posteriormente, enviar sus credenciales (puede ser un nombre de usuario y contraseña cifrada, pseudónimo, etc.) al NAS. Este elemento intermedio envía un mensaje RADIUS de petición de acceso, *Access Request*, al servidor RADIUS mediante el cual solicita autorización para el acceso al recurso o servicio. Durante este proceso, por norma general, se establecerá con el servidor el mecanismo de autenticación que se utilizará para llevar a cabo el intercambio de las credenciales del usuario y el proceso de autenticación en general. Dichas credenciales suelen ser un nombre de usuario y contraseña (protegida mediante algún mecanismo

que proporcione seguridad), o un certificado de cliente proporcionado por el usuario con sus credenciales. Además, esta solicitud puede incorporar más información que conoce el NAS sobre el usuario, como dirección de red, información del punto físico de conexión del usuario, etc.

Es entonces cuando el servidor RADIUS recibe el mensaje *Access Request* y procede a verificar si la petición recibida es correcta siguiendo el modo de autenticación elegido (PAP [5], CHAP [6], EAP [7], etc.). Verifica los datos proporcionados por el usuario con los almacenados por el servidor, ya sea en una base de datos en texto plano, un directorio LDAP [15] o una base de datos MySQL [16]. En este punto, el servidor RADIUS puede devolver tres posibles respuestas en función de si ha sido correcta la autenticación, errónea, o si necesita más información para conceder el acceso o no.

- *Access Reject*. El servidor envía este mensaje al NAS (cliente RADIUS) si el acceso al usuario es denegado incondicionalmente a todos los recursos o servicios solicitados. Como causas podemos tener la falta de datos proporcionados por el cliente, que se trate de una cuenta o un usuario desconocido para el servidor RADIUS, o un fallo en el proceso de autenticación.

- *Access Challenge*. Este mensaje es enviado por el servidor RADIUS cuando se quiere solicitar al usuario que aporte información adicional para poder llevar a cabo su autenticación, como información adicional de autenticación, PIN, token, etc. También se utilizan estos mensajes para modos de autenticación más complejos que establecen un túnel seguro para poder enviar las credenciales de acceso para que no sean visibles a elementos intermedios.

- *Access Accept*. Con este mensaje se le concede el servicio o recurso al usuario. Una vez que ha sido autenticado y verificada su identidad, el servidor RADIUS procede a verificar si tiene autorización al servicio o recurso solicitado, por ejemplo, el acceso a una red inalámbrica. Esta información de autorización puede estar almacenada en el mismo servidor, en un servidor LDAP o una base de datos, que consultará el servidor RADIUS.

Cada una de estas tres posibles respuestas puede incluir un atributo *Reply-Message* que puede contener una razón por la cual se denegó el acceso, información útil para el *Access Challenge* o un mensaje de bienvenida en el caso en el que se envíe un *Access Accept*. En este último mensaje (*Access Accept*) también se pueden añadir atributos de autorización, como por ejemplo la dirección IP específica concedida al *usuario*, parámetros de QoS (calidad de servicio), ajustes VLAN [17], el tiempo máximo que el usuario puede permanecer en la red o utilizando el servicio, etc. Más adelante ampliaremos la información relativa a los atributos.

En la **Figura 1**, podremos ver un ejemplo del flujo mencionado anteriormente.

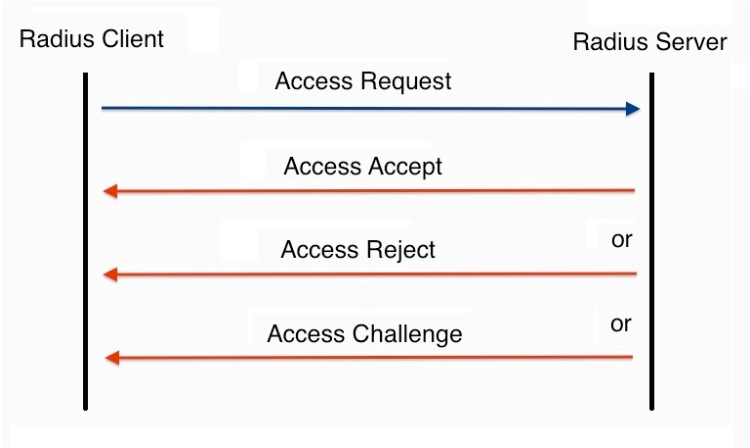

Figura 1: Intercambio básico de mensajes RADIUS.

Seguidamente, en la **Figura 2**, podemos ver un esquema del funcionamiento básico en la autenticación de un cliente con un servidor RADIUS.

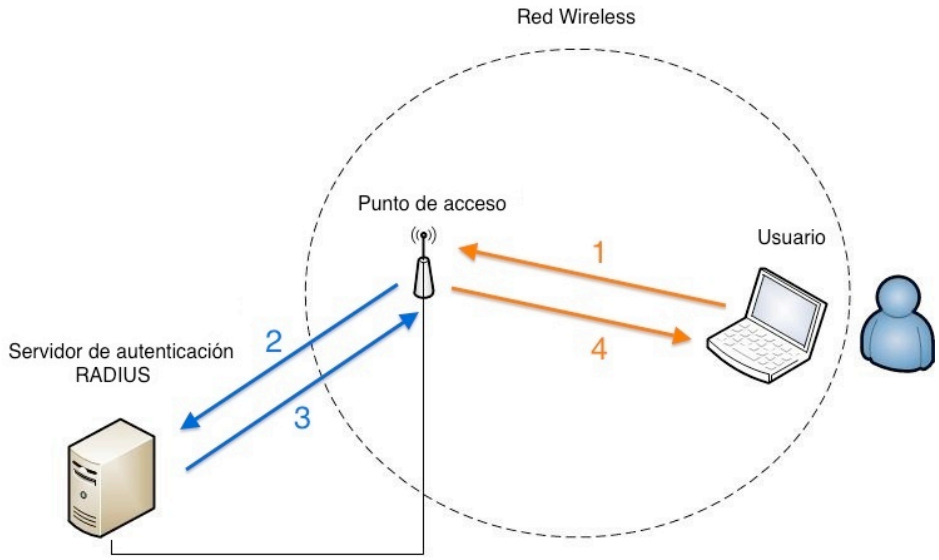

Figura 2: Escenario básico de autenticación con RADIUS.

Vemos como el usuario quiere autenticarse en una red WiFi y para ello el punto de acceso (NAS) enviará un *Access Request*. Éste se comunica con el servidor RADIUS que le dirá si la autenticación es correcta o no y en función de lo recibido, el punto de acceso permitirá el acceso a la red Wifi o no. El protocolo RADIUS AAA se intercambia en los pasos 2 y 3, junto con los mensajes EAP para autenticar al usuario, que irán encapsulados dentro de los paquetes RADIUS. Por otro lado, entre el NAS y el cliente también viajan los mensajes EAP, pero esta vez encapsulados en el protocolo EAPOL [21]. Hay que dejar claro, que la autenticación EAP se realiza entre cliente y servidor RADIUS, actuando como cliente RADIUS el NAS.

Otra característica que proporciona un protocolo AAA como RADIUS es el soporte de roaming entre ISPs (Proveedor de Servicios de red). En primer lugar, el roaming entre *ISPs* va a permitir la autenticación de un usuario de un dominio en un *ISP* que es *distinto* al que pertenece el usuario. De esta forma, se garantiza que un usuario, se encuentre donde se encuentre, podrá ser autenticado y autorizado a utilizar un recurso o servicio.

RADIUS facilita este uso gracias a los *realms*. Estos *realms* se añaden al nombre de usuario, delimitado por @. De esta forma, podemos saber a qué dominio pertenece el usuario y encaminar la petición hacia esos servidores. En la siguiente imagen, **Figura 3**, podemos ver un ejemplo del funcionamiento del roaming.

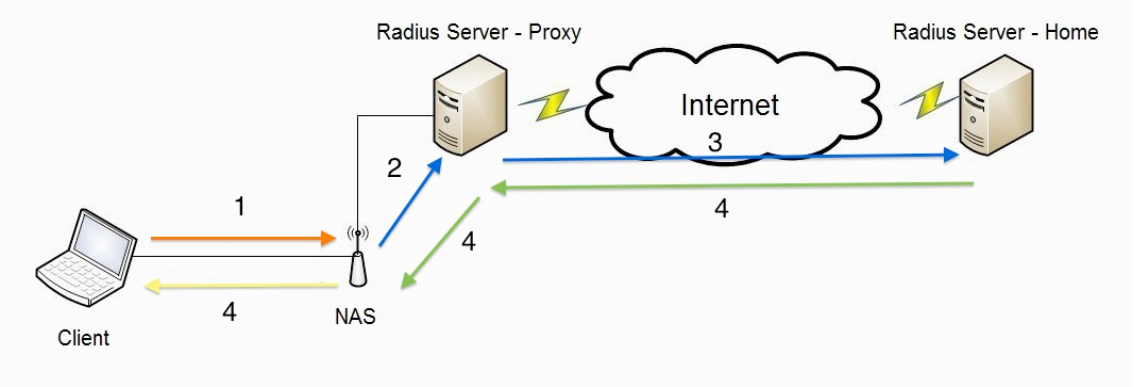

Figura 3: Proceso de roaming RADIUS.

En el paso 1, el cliente quiere autenticarse y envía la petición de acceso a la red al NAS. A continuación, en el paso 2 el NAS envía un paquete *Access Request* a su servidor RADIUS local con la información del usuario. Entonces, el servidor ve que pertenece a un dominio que él no puede procesar y la encamina a través de la infraestructura AAA, como podemos ver en el paso 3, va al servidor "home" del usuario. Es aquí cuando el servidor RADIUS procesa la petición de autenticación y la envía de vuelta hasta el cliente, que es el paso 4.

2.2.2 Estructura del paquete

Un paquete RADIUS consta de una cabecera con información relativa al paquete como:

- Código. Identifica el tipo de paquete, indicando si es un *Access Request* (1), *Access Accept* (2), *Access Reject* (3), *Access Challenge* (11), etc.
- Identificador de paquete. Utilizado para asociar peticiones con respuestas y evitar ataques de reenvío.
- Longitud. Indica la longitud completa del paquete RADIUS, incluida la cabecera. La longitud máxima que puede tener un paquete es de 4096 bytes, lo que puede provocar problemas a la hora de enviar información de gran tamaño en los paquetes. Este aspecto es de especial importancia en este TFG, tal y como veremos.
- Autenticador. Este campo se utiliza para autenticar la respuesta del servidor RADIUS y también para *proteger las contraseñas* que pueden ser enviadas en los paquetes.
- Pares atributo valor (AVP). El resto del espacio que queda libre en el paquete se destina al envío de estos pares atributo valor. Estos atributos transportan datos tanto en las solicitudes como en las respuestas de autenticación, autorización y accounting. Son muy importantes, siendo la base del protocolo, ya que son los encargados de transportar la información del usuario para poder verificar su identidad y proporcionar el recurso o servicio que solicita, así como para enviar el resultado y los datos que indiquen el resultado del proceso. También son muy útiles a la hora de enviar los datos de accounting, para tarificar y saber el uso del

usuario de los recursos. Los atributos se definen del tipo par atributo valor, y dicha estructura podemos apreciarla en la siguiente imagen, **Figura 4**.

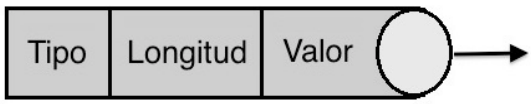

Figura 4: Estructura atributo RADIUS.

Observamos como un AVP consta de un campo *tipo*, que nos dice el tipo de atributo que transportamos; otro campo que indica la longitud del mismo; y el campo que contiene el valor del atributo. Como máximo, un AVP, podrá tener una longitud de 255 bytes.

Ejemplos de atributos enviados en un paquete RADIUS:

1. User-Name: type 1.
2. User-Password: type 2.
3. Service-Type: type 6.
4. Reply-Message: type 18.
5. State: type 24.
6. Vendor-Specific: type 26.
7. Proxy-State: type 33.
8. EAP-Message 79.
9. Etc.

También, podemos definir nuevos atributos. Esto es una característica de la extensibilidad que aporta RADIUS. Por ejemplo, proveedores de hardware, o simplemente en *determinados escenarios es conveniente* definir nuevos atributos o posibles valores a algunos ya existentes. Para poder añadir estos nuevos atributos se hace uso del atributo 26 (Vendor-Specific), donde se indica que vamos a utilizar un atributo que no pertenece al protocolo y es propio de un diccionario específico, de un proveedor hardware o de otro escenario, y por lo tanto será encapsulado con su número correspondiente dentro del atributo Vendor-Specific.

2.2.3 Seguridad

RADIUS también aporta unos aspectos de seguridad básicos. Por ejemplo, las contraseñas de los usuarios nunca viajan en claro entre el NAS (cliente RADIUS) y el servidor RADIUS (con PAP es opcional). Siempre, para transmitirlas, como mínimo se hace un hash MD5 [18] con secreto compartido entre ambas entidades. Aún así, para dar mayor protección a las credenciales se puede optar por añadir protección adicional como túneles IPSec [19] o TLS [20]. A su vez, existe el atributo Message-Authenticator que proporciona protección y seguridad a todos los paquetes que lo incorporen. Por otro lado, diversos atributos sensibles que van en los paquetes RADIUS, como identificadores de túneles o identificadores de VLAN, van en claro y no son protegidos por el propio protocolo RADIUS, por lo que puede ser necesario utilizar alguno de los túneles antes descritos. El protocolo RadSec [32], que es una variante de RADIUS aporta más seguridad y pretende minimizar los problemas de seguridad que hemos descrito. En particular, se envían los mensajes sobre un canal TCP protegido con TLS.

También se establece entre los servidores y clientes RADIUS una relación de confianza, la cual permite autenticar las peticiones y respuesta recibidas. Se consigue gracias a secretos compartidos establecidos entre estas entidades.

Después de ver los aspectos generales y las características de RADIUS podemos entender que presenta varios problemas. Algunos de ellos son la gestión de solicitudes simultáneas (255 máximo), limitación en el tamaño de los paquetes (4096 bytes, es el problema que queremos resolver), los tamaños de los campos dentro de un mensaje, etc.

2.2.4 RADIUS y EAP

Para finalizar, sería conveniente establecer una relación entre RADIUS y EAP. Como hemos comentado antes, RADIUS permite múltiples métodos de autenticación. Por ejemplo, podemos utilizar CHAP como método de autenticación. CHAP permite autenticar a un usuario mediante desafío-respuesta. Lo realiza verificando la identidad del usuario mediante un intercambio de tres etapas. De esta forma, implica que RADIUS conozca el mecanismo de autenticación que se utiliza, la forma de autenticar de cada uno, peculiaridades, etc.

Sin embargo, EAP permite abstraer a RADIUS del método de autenticación que se utilice. Proporciona una capa genérica a RADIUS mediante la cual, un usuario puede ser autenticado haciendo uso de cualquier mecanismo de autenticación que irá encapsulado dentro de EAP. De esta forma, para autenticar a un usuario se utilizaría EAP-CHAP, EAP-MD5, etc. Y si alguno de los métodos de autenticación cambian (CHAP, MD5, LEAP) no sería necesario actualizar el servidor RADIUS, simplemente tendríamos que actualizar el módulo EAP del servidor. Por esta razón, es muy común utilizar EAP como método de autenticación, que encapsulará al mecanismo que seguirán ambas partes para llevar a cabo la autenticación.

Todo el intercambio EAP entre cliente AAA y servidor AAA se llevará a cabo mediante los paquetes RADIUS explicados anteriormente.

2.3 EAP

EAP (Extensible Authentication Protocol) [7] es un protocolo genérico de autenticación que proporciona transporte para los mensajes de autenticación generados por los denominados métodos EAP. Cada método define un mecanismo de autenticación diferente. Cada mecanismo de autenticación que utiliza EAP (método EAP) define una forma de encapsular sus mensajes dentro de EAP. Por lo que, como hemos dicho, es un marco de autenticación, no un método de autenticación específico, aunque EAP sí proporciona algunas funciones y negociaciones comunes a todos los métodos EAP. Actualmente es muy usado tanto en redes Wireless como en redes cableadas.

En un escenario típico de autenticación tendríamos tres entidades bien diferenciadas, EAP Peer, EAP Authenticator y EAP Server. Explicadas en detalle y mapeadas con las ya vistas en RADIUS serían:

- *EAP Peer*. Se corresponde con el cliente que quiere acceder al servicio. Es quien solicita la autenticación, el software del cliente. Se comunica con el *EAP Authenticator* mediante un EAP lower-layer (p. ej. EAPOL) [21].

- *EAP Authenticator*. Se trata del elemento intermedio que se encarga de conceder el acceso al servicio solicitado por *EAP Peer*. En RADIUS es el NAS.

- *EAP Server*. Se corresponde con el servidor AAA que autenticará al usuario. Es quien recibe las peticiones EAP y autoriza o no al usuario para utilizar el servicio.

El funcionamiento de EAP se distribuye generalmente en 4 pasos:

1. El *EAP Authenticator* (NAS) envía un *EAP Request ID* al *EAP Peer*. Dicho mensaje sirve para solicitar la identidad del EAP Peer.

2. El cliente (EAP Peer) envía un paquete *EAP Response ID* con su identidad al *EAP Authenticator,* el cual es reenviado hacia el *EAP Server* encapsulado en un mensaje RADIUS para iniciar el proceso de autenticación. Al igual que en el paquete *EAP Request ID*, este paquete contiene un campo de *type*, el cual tiene que tener el mismo valor del campo type en el paquete *EAP Request ID*.

3. El *EAP Server* responde a este mensaje recibido encapsulando la información EAP dentro de RADIUS. Esta información la recibe el *EAP Authenticator* y envía un *EAP Request* con el contenido recibido, al cual el EAP Peer responde con un *EAP Response*. La secuencia de Request y Response entre *EAP Peer* y *EAP Server* continúa mientras sea necesario según el mecanismo EAP que se va a utilizar para autenticar. Estos mensajes EAP permiten que el usuario pueda demostrar su identidad y enviar la información que sea necesaria para que el *EAP Server lo valide*. Hay que tener en cuenta que EAP es un protocolo lock-step, por lo que no se puede enviar el siguiente paquete sin haber recibido uno válido antes. Esto implica que el *EAP Server* sea responsable de retransmitir los paquetes que se hayan podido perder.

4. Los mensajes descritos en el punto anterior seguirán hasta que el EAP Server no pueda autenticar al cliente, donde enviará un *EAP-Failure*, o hasta que se haya producido una autenticación satisfactoria, donde enviará un paquete *EAP-Success*. En este caso el *EAP Authenticator* y el *EAP Peer* pueden negociar una clave que podrá ser utilizada, por ejemplo, para establecer conexiones cifradas.

A continuación podemos ver la **Figura 5** en la que se muestra el intercambio de mensajes que se lleva a cabo cuando se realiza una autenticación utilizando EAP. En dicha figura podemos observar una autenticación utilizando las tres entidades EAP descritas anteriormente. Hay que destacar que EAP viaja desde el cliente hasta el servidor (*EAP Server*, que en este caso es un servidor RADIUS). Entre el *EAP Peer* y el *EAP Authenticator*, EAP viajará sobre un EAP lower-layer como EAPOL [21]; y entre el *EAP Authenticator* y el *EAP Server* viajará dentro de los paquetes RADIUS que explicamos anteriormente.

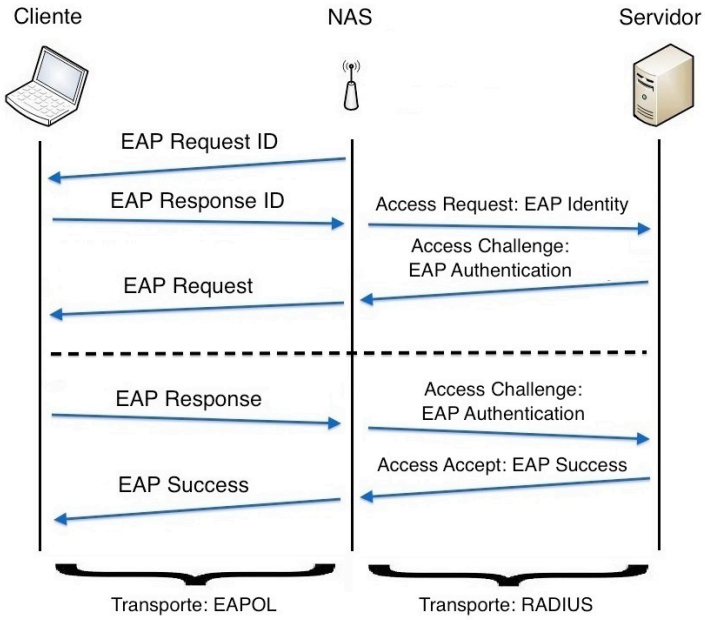

Figura 5: Flujo de autenticación EAP mediante RADIUS.

Anteriormente hemos hablado de mecanismos que son los que realmente llevan a cabo el intercambio necesario para el proceso de autenticación del usuario. Estos mecanismos son los llamados métodos EAP. Ejemplos de dichos protocolos son EAP-MD5 [22], EAP-SIM [23], EAP-TLS [33], EAP-TTLS [4], EAP-PAP [7], etc. Difieren entre ellos en la forma en la que autentican. Por ejemplo, EAP-MD5 no es más que un hash de la clave del usuario para autenticarle. Es inseguro [7] siendo vulnerable a múltiples ataques y teniendo carencias como la ausencia de autenticación mutua entre cliente y servidor [22]. Por otro lado también tenemos EAP-TTLS, que se basa en el establecimiento de un túnel TLS, para que dentro de él *se túnele otro mecanismo EAP*, como EAP-MD5, dando mas seguridad.

Los métodos más utilizados son EAP-TLS, EAP-SIM, EAP-AKA, PEAP, LEAP y EAP-TTLS. Los requerimientos de dichos métodos en redes inalámbricas están descritos en el RFC 4017 [34]. Cuando EAP es invocado por un dispositivo NAS (Network Access Server) capacitado para 802.1X (WPA Enterprise) [24], como por ejemplo un punto de acceso inalámbrico 802.11 (WiFi), los métodos EAP antes indicados proveen un mecanismo seguro de autenticación y negocian una PMK (Pair-wise Master Key) entre el dispositivo cliente y el NAS. En esas circunstancias, la PMK puede ser usada para abrir una sesión inalámbrica cifrada que use cifrado TKIP o AES. La gestión de estas claves puede verse más en detalle en el RFC 4017 [34].

2.4 SAML

SAML (Security Assertion Markup Language) [9] es un lenguaje basado en XML [25] para el intercambio de información de autenticación y autorización. Habitualmente es intercambiado en escenarios de Single Sign-On (SSO) [1], generalmente entre un Proveedor de Identidad (IdP) y un Proveedor de Servicio (SP). Creado por el comité OASIS [35] en 2001, la primer versión está actualmente en desuso. En 2005 se ratificó como estándar OASIS la versión 2 [9] del protocolo, siendo la estable y usada actualmente, reemplazando a la versión 1.1. Los aspectos más importantes de SAML

están recogidos en diversos documentos, como SAMLConform [39], SAMLCore [36], SAMLBind [37] y SAMLProf [38].

El principal objetivo de SAML es hacer posible que exista tanto una federación de identidad como SSO. Una federación de identidad permite a un SP delegar el proceso de autenticación a una entidad externa y diferente, como puede ser un IdP. Por otro lado, SSO permite que un usuario pueda acceder a diferentes SPs con un sólo proceso de autenticación. Esta cualidad es posible mientras que el tiempo de vida del token de autenticación que permite el SSO no haya expirado. Dicho token, puede ser un token basado en SAML, que puede dar las condiciones e información necesaria para que se produzca una federación de identidad y SSO.

Las entidades que forman parte de un escenario SAML son las siguientes:

- Principal. Normalmente será el usuario. La entidad que quiere acceder a un servicio o recurso.

- Proveedor de Identidad (IdP). Entidad que se encarga de verificar la identidad del usuario y generar una respuesta a su solicitud. Es la que tiene acceso a las credenciales del usuario y las compara con las que le presenta en la solicitud.

- Proveedor de Servicio (SP). Es la entidad que se encarga de recibir la petición del rol principal, redirigirla al IdP para que la procese y en base a lo que le conteste autorizar o no el acceso al recurso o servicio que él proporciona.

La forma de funcionamiento del escenario básico que utiliza SAML es la siguiente. La entidad principal solicita un servicio al proveedor de servicios (SP). Éste a su vez solicita al proveedor de identidad (IdP) que verifique la identidad del usuario y genere una respuesta para el recurso o servicio solicitado. En caso de éxito, el proveedor de servicio obtiene una confirmación de identidad. Teniendo como base esta confirmación recibida, el proveedor de servicio puede tomar decisiones acerca del acceso autorizado a un usuario al recurso o servicio que él ofrece. En el caso en el que el usuario no haya sido exitosamente autenticado se generará un error, mediante el cual no tendrá acceso al servicio o recurso que ha solicitado.

Un escenario donde encontramos SAML, identidad federada y SSO es el perfil Web-SSO. En este caso, aplicamos SSO sólo con aplicaciones o recursos accesibles vía web. Se detecta a los usuarios no autenticados para redirigirlos a un proveedor de identidad (IdP), regresando al recurso o servicio cuando haya finalizado el proceso de autenticación para que el SP autorice o deniegue el acceso en caso de error. Se utilizan cookies [40], que nos permitirán reconocer a los usuarios que ya han accedido y cual es el estado de su autenticación.

El uso de SAML dentro de este tipo de SSO, sería a la hora de redirigir al usuario al IdP y en la vuelta desde el IdP al SP. En el primer caso, estaríamos enviando un *SAMLAuthenticationRequest* y en segundo lugar enviaríamos un *SAMLResponse*, que contiene uno o más *Assertions*, y dentro de ellos los *Statements* que contienen la información acerca de la autenticación, atributos del usuario, roles, permisos, etc. En la **Figura 6** podemos ver un ejemplo de este escenario.

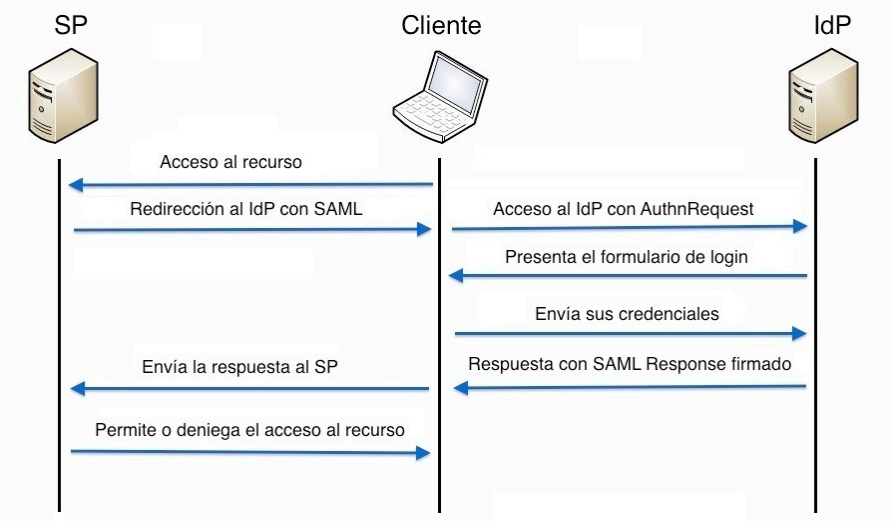

Figura 6: Escenario Web-SSO con SAML.

Por lo tanto, el intercambio de información de identidad se realiza mediante los mensajes *SAMLAuthenticationRequest* y *SAMLResponse* que proporciona el protocolo SAML. Estos mensajes permiten el envío de solicitudes y respuestas con información. Un *SAMLAuthenticationRequest* contiene un *AuthnRequest* mediante el cual el SP solicita al IdP que autentique al usuario que lo representa. Esta petición puede ir firmada y también suele llevar información acerca del mecanismo preferido de autenticación. Un *SAMLResponse* contiene información para que el proveedor de servicio pueda conceder el acceso al recurso o no.

Dentro de ese *SAMLResponse* normalmente irán sentencias llamadas *Assertions*. Los *Assertions* contienen información formada por cero o más *Statements* creados por una autoridad (IdP). Por lo que los *Assertions* son realmente datos acerca de ciertos hechos que van siempre relacionados con una entidad, los cuales permiten tomar decisiones y conocer información acerca del usuario. En términos generales, una *sentencia* interpretada por el SP (cliente que espera el *SAMLResponse*) viene a decir que dicho *Assertion* se expidió en el *instante t* por el *emisor R* en referencia al *sujeto S* proporcionando las *condiciones C y que son válidas*. Dicho de otra forma, indica la información que es válida emitida por un IdP en un tiempo determinado.

Los *Assertions,* como hemos comentado, pueden contener *Statements*, creadas todas ellas por una misma autoridad. Encontramos tres tipos diferentes de *Statements*, los cuales están todos definidos en base a un mismo *Subject*, que identifica al usuario.

- Statement de autenticación (AuthStatement). Indica que el usuario fue autenticado en un momento particular por un medio usando un método particular de autenticación.

- Statement de atributo (AttributeStatement). Indica atributos acerca del usuario.

- Statement de autorización (AuthzDecisionStatement). Indica si el usuario tiene acceso garantizado o rechazado al recurso pedido. Básicamente se dice que el usuario U tiene permitido realizar la acción A sobre el recurso R dada la evidencia E, que puede ser una autenticación correcta.

Además de estos *Statements* una sentencia puede contener más información. Por ejemplo, hemos comentado antes que todo lo que se encuentra dentro de un *Assertion* va definido en base a un mismo *Subject*. Este S*ubject* hace referencia a la parte a autenticar, es decir, al usuario. Es por ello, que todos los S*tatements* dentro de un *Assertion* se refieren al mismo usuario para dar información acerca del mismo. Por lo tanto, uno de los campos que puede tener también un *Assertion* SAML es el *Subject*. Otros campos que también podemos encontrar son el *issuer*, que contiene el identificador del proveedor de identidad (IdP), *Signature* que contiene una firma digital del *Assertion* completo para preservar su integridad, *Conditions* que indica en que condiciones debe considerarse válido el *Assertion*, etc.

Para verlo más claro, en la **Figura 7** podemos observar el contenido de un mensaje *SAMLAuthenticationRequest* y un *SAMLResponse*, y dentro de él las partes que podemos encontrar ya explicadas, como el *Assertion* y los *Statements*.

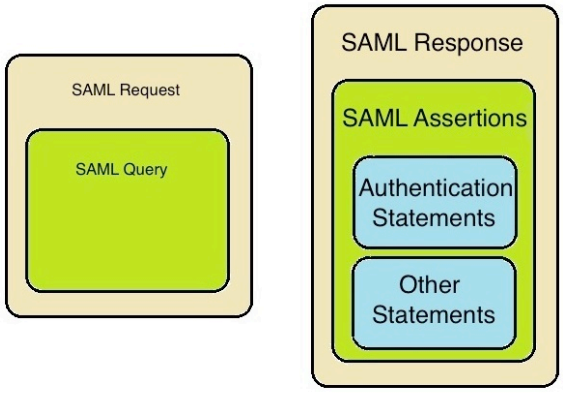

Figura 7: Estructura mensajes SAML.

SAML también recomiendan algunos aspectos de seguridad. Por ejemplo el uso de SSL para el transporte seguro de las sentencias entre entidades, o de XMLSignature [26] y XMLEncryption [27] para la seguridad a nivel de mensaje. Centrándonos en estos dos últimos, el primero nos permite firmar un mensaje SAML y el segundo nos permite cifrar un mensaje. Más en detalle, XMLSignature define una *sintáxis XML* para la firma digital. Está principalmente orientado para la firma de documentos definidos en XML, como es el caso de SAML, aunque puede ser válido para firmar cualquier contenido.

Un ejemplo de un *Assertion* firmado con XMLSignature se puede observar en la **Figura 8**. En él podemos observar la firma de un *Assertion* con un *AuthStatement*.

```
<samlp:Response>
    ...
    <saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema" ID="pfx8377c4fd-57f1-be08-2bfd-6223ad8dc87b"
        Version="2.0" IssueInstant="2014-04-30T21:17:31Z">
        <saml:Issuer>http://155.54.225.6/simplesamlphp/saml2/idp/metadata.php</saml:Issuer>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <ds:SignedInfo>
                <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
                <ds:Reference URI="#pfx8377c4fd-57f1-be08-2bfd-6223ad8dc87b">
                    <ds:Transforms>
                        <ds:Transform
                            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
                        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                    </ds:Transforms>
                    <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                    <ds:DigestValue>7x4XLB69lfq8n6D30ora/7s9uaI=</ds:DigestValue>
                </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>SU64A7AonKYkLolOZkx...</ds:SignatureValue>
            <ds:KeyInfo>
                <ds:X509Data>
                    <ds:X509Certificate>MIID5TCCAs2gAwIBdzc...</ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </ds:Signature>
        <saml:Subject> ... </saml:Subject>
        <saml:Conditions NotBefore="2014-04-30T21:17:01Z"
            NotOnOrAfter="2014-04-30T21:22:31Z">
            <saml:AudienceRestriction>
                <saml:Audience>http://155.54.225.5/simplesamlphp/module.php/saml/sp/metadata.php/default-sp</saml:Audience>
            </saml:AudienceRestriction>
        </saml:Conditions>
        <saml:AuthnStatement AuthnInstant="2014-04-30T21:17:31Z"
            SessionNotOnOrAfter="2014-05-01T05:17:31Z" SessionIndex="_94acbd1a5458d4cab4dac468639eb6966a3aab8eb1">
            <saml:AuthnContext>
                <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</saml:AuthnContextClassRef>
            </saml:AuthnContext>
        </saml:AuthnStatement>
    </saml:Assertion>
</samlp:Response>
```

Figura 8: Ejemplo de XML Signature

Por lo que podemos apreciar, si estamos realizando una firma de un *Assertion* SAML con una cantidad importante de *Statements*, podríamos encontrarnos con la necesidad de enviar un gran *SAMLRequest* o *SAMLResponse*. De este modo, un *Assertion* típico que incluya una sentencia de autenticación y una de atributos, firmado digitalmente y que incluya un certificado digital puede llegar a ocupar un tamaño aproximado de 15 KiloBytes [44].

Podemos concluir afirmando que SAML aporta a la hora de autenticar a un usuario la capacidad de poder dar más información que una simple afirmación de si se está autenticado o no. Por ejemplo, decidir qué para ciertos recursos si se podrá tener permiso, para otros no, en qué condiciones e incluso aportar información más significativa del usuario, como sus atributos o roles.

2.5 ABFAB/Moonshot

ABFAB (Application Bridging for Federated Access Beyond web) es un grupo de trabajo del IETF [41] encargado de definir una arquitectura para proporcionar acceso federado a servicios no basados en Web (p. ej. Telnet, FTP, etc.). A su vez, Moonshot es la comunidad de trabajo que está implementando los estándares definidos en el grupo de trabajo ABFAB. Dicha arquitectura hace uso de extensiones de los mecanismos de seguridad de uso común, tanto para federados como para no federados, como por ejemplo usando RADIUS, GSS-API [29], EAP y SAML.

En primer lugar hay que describir que es un acceso federado. Como vimos en la sección 2.4, acceso federado significa que la entidad que autentica (IdP) y la que tiene el recurso (SP) son diferentes. Por lo que existe una federación entre estas dos entidades diferentes que pueden pertenecer o no, a la misma organización. A su vez,

las tecnologías que utilizan un acceso federado pueden hacer uso de SAML para el envío de la información relacionada con el cliente, al igual que ABFAB/Moonshot. Pero para poder tener un acceso federado, es necesario tener un conjunto de prácticas y protocolos entre las entidades que formen la federación de identidad.

En ABFAB/Moonshot, dentro de un acceso federado, interactúan tres partes, un cliente, un IdP y el Relying Party (RP). Todos los hemos explicado antes, correspondiéndose el RP con la entidad que ofrece el servicio en el proveedor de servicio (SP). Por lo tanto el RP es cliente de un IdP, y su función es obtener/verificar las credenciales de seguridad de un usuario. Por lo tanto, un RP confía en IdPs para llevar a cabo la autenticación de un usuario, conformando el acceso federado.

En ABFAB/Moonshot se conecta el RP (que es un servicio no web) con la infraestructura AAA, delegando a dicha infraestructura la autenticación y autorización, permitiendo dicho acceso federado.

Como contrapartida, realizar esta función y delegar en AAA para permitir el acceso federado tiene unos problemas. En primer lugar, ahora el servicio tiene que incorporar el rol de cliente RADIUS. Es decir, tiene que implementar la funcionalidad AAA para poder autenticar a los usuarios que quieran utilizar dicho servicio.

En segundo lugar, como se dijo anteriormente, lo más beneficioso es utilizar en RADIUS el método de autenticación EAP y que éste llevara encapsulados los mecanismos reales de autenticación. De esta forma abstraemos a RADIUS del verdadero mecanismo de autenticación. En este caso, si en ABFAB/Moonshot necesitamos implementar un cliente RADIUS y queremos utilizar EAP como método de autenticación genérico, ¿cómo llevamos EAP desde el cliente de aplicación al servicio que va a autenticarse mediante RADIUS? Para ello, existe una solución llamada GSS-API [29], que tiene como finalidad aportar una API que permite usar sistemas de seguridad de forma genérica. Dentro de ABFAB/Moonshot se ha reutilizado GSS-API para permitir transportar datos de autenticación y autorización desde el cliente hasta el servicio. Aporta una interfaz con operaciones entre estas dos entidades para llevar el acceso federado mediante RADIUS a distintos servicios de aplicación . En particular ABFAB/Moonshot propone transportar EAP sobre GSS-API, surgiendo así GSS-EAP [51]. De este modo, ABFAB/Moonshot utiliza este método para llevar EAP desde el cliente de la aplicación hasta el RP, y posteriormente utilizaremos RADIUS entre el RP y el IdP. Así dotamos de autenticación y autorización federada a servicios que no la tienen a través de una federación AAA.

Por último, queda un problema por resolver. Básicamente, queremos autenticar y autorizar a un usuario pero, ¿cómo gestionamos esa autorización? Generalmente se utiliza SAML para gestionar y transportar dicha información de autorización. En ABFAB/Moonshot podría existir la necesidad del envío de datos pre-autorización y de post-autorización. Estos datos están definidos en SAML. Los primeros datos de autorización serían utilizados durante el proceso de autenticación y los segundos serían enviados hacia el cliente con información acerca de su autenticación, perfil, permisos, roles, etc. El problema es que ahora vamos a utilizar RADIUS para transportar SAML [9]. Por tanto, es necesario definir como llevar a cabo este transporte. En ABFAB/Moonshot se consigue creando un diccionario que contiene atributos definidos específicamente para transportar SAML.

En resumen, las tecnologías que forman la arquitectura ABFAB/Moonshot son:

- GSS-API entre la aplicación de cliente y RP.

- RADIUS entre RP e IdP.
- EAP entre la aplicación de cliente y el IdP.
- SAML para la representación de la información de autorización.

En la **Figura 9** podemos ver una muestra del flujo llevado a cabo en ABFAB/Moonshot para la autenticación.

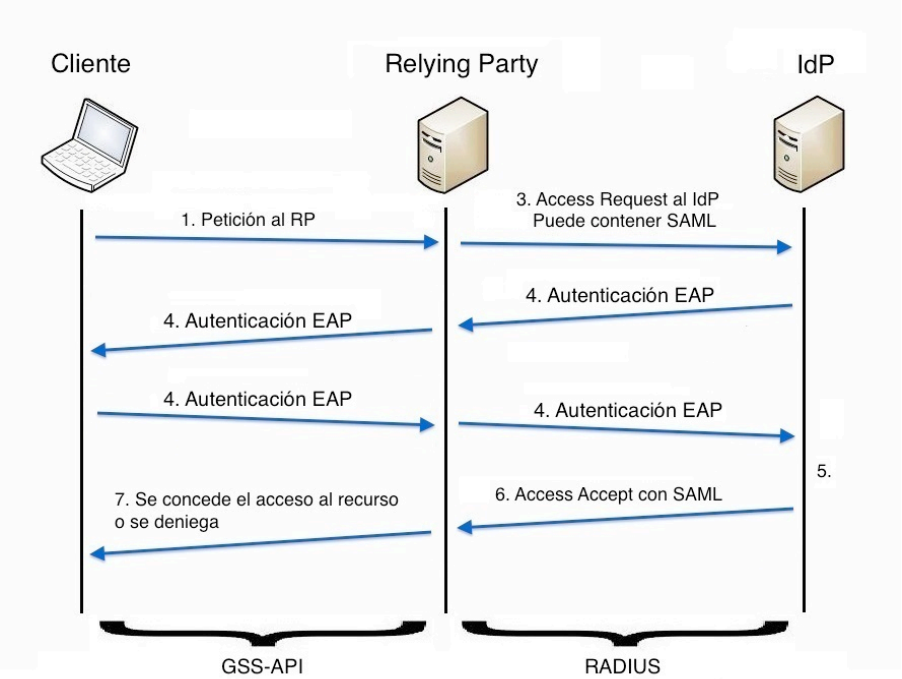

Figura 9: Flujo de autenticación en ABFAB/Moonshot con SAML

A continuación tenemos una explicación de lo ocurrido en cada punto.

1. El cliente inicializa los datos necesarios para la autenticación. Para ello configura la identidad a presentar durante el proceso de autenticación, elije el mecanismo GSS-EAP para autenticar al usuario y transmite esta información al RP para iniciar el proceso de autenticación.

2. El siguiente paso es descubrir donde se encuentra el IdP. Esto lo realizará el RP, ya que esta entidad es la que conoce quien gestiona la autenticación y proporciona el acceso federado.

3. Una vez encontrado, el RP envía un *Access Request* (RADIUS) al IdP para iniciar el proceso de autenticación federado. Dicho paquete puede contener datos de autorización útiles para el proceso de autenticación (datos de pre-autorización). Por ejemplo podría tratarse de datos SAML de gran tamaño.

4. En este punto, el IdP hablará directamente con el cliente para establecer el método EAP a utilizar y para proceder al intercambio de información para autenticar al cliente. Se realizará con sucesivos mensajes RADIUS Access Request y Access Challenge entre el RP y el IdP.

5. Una vez autenticado, el IdP comprobará los permisos del usuario para conceder la autorización al servicio.

6. Si la autenticación y la autorización han ido exitosas, el IdP envía un Access Accept hacia el RP. Dentro de éste paquete RADIUS podría ir incluido un *Assertion* SAML que proporcione información adicional de autorización al cliente (datos de post-autorización). Dicho *Assertion* SAML podría ser de gran tamaño.

7. Por último el RP procesa el paquete recibido y el contenido del *Assertion* para enviar el resultado al cliente.

El envío de un *Assertion* SAML es totalmente opcional, pudiendo ser necesario debido a que el RP puede que necesite más datos, además de una simple aceptación por parte del IdP, como atributos, roles, etc… Estos datos contenidos dentro del *Assertion* ayudarán a tomar una decisión y determinar si el acceso es permitido o no. Debido a esto, será necesario utilizar IdPs que permitan el procesamiento de SAML, al igual que los RPs. A su vez, debido a que dichas sentencias SAML generalmente podrían ser grandes, será necesario que el cliente y servidor RADIUS sean capaces de soportar fragmentación RADIUS.

3. Análisis de objetivos y metodologías

Como se ha comentado anteriormente, este trabajo se basa en la propuesta definida en [31], que intenta resolver la problemática de poder enviar datos tanto de pre-autorización como de post-autorización de tamaño superior a 4096 bytes en un proceso de autenticación y autorización basado en RADIUS. La razón se debe a que este protocolo sólo permite este tamaño máximo para un paquete. La solución planteada en esta propuesta actualmente es un borrador (draft) del IETF. Si finalmente es aprobado, pasará a ser un RFC experimental. Dicha solución es la de fragmentar los datos en varios intercambios RADIUS que no superan el límite del protocolo, de forma que en un número finito de mensajes se pueda recibir y reconstruir el paquete original en el receptor.

En base a esto, la realización de este TFG tiene unos objetivos determinados.

- Implementar toda la funcionalidad, flujos y entidades que define la propuesta de fragmentación.

- Ofrecer una implementación de código abierto.

- Validar la implementación con casos de uso realistas. Para ello se integrará la funcionalidad dentro de la arquitectura definida en ABFAB/Moonshot, anteriormente mencionada.

Por último, para la realización del trabajo se han seguido los siguientes pasos:

- Leer toda la documentación acerca de la propuesta que contiene los aspectos de diseño de la fragmentación para RADIUS.

- Diseñar las estructuras y métodos que vamos a necesitar implementar para integrar la funcionalidad de fragmentación.

- Llevar a cabo la implementación de la propuesta creando las estructuras de datos y las funciones necesarias en el servidor.

- Implementar un cliente básico que entienda cuando un flujo necesita soporte de fragmentación, tanto de pre-autorización como de post-autorización, añadiendo las funciones y estructuras necesarias para ello.

- Una vez que en el cliente básico anterior se compruebe que la especificación y el soporte de fragmentación realizado funciona correctamente, se llevará a cabo la integración en ABFAB/Moonshot. Los pasos son:

    - Conocer en detalle la arquitectura propuesta por ABFAB/Moonshot y comprobar el flujo establecido por las entidades cuando se realiza una autenticación a través de ABFAB/Moonshot.

    - Integrar en el software de ABFAB/Moonshot la funcionalidad de fragmentación. Para ello, de forma muy similar al *cliente básico* anterior, se implementarán las funciones y estructuras necesarias para el soporte de fragmentación.

4. Diseño e implementación de la solución propuesta

En este punto vamos a explicar en detalle el desarrollo de la solución, y cómo se ha llevado a cabo el diseño y la implementación de la misma.

Como se ha comentado anteriormente, este trabajo se basa en el diseño y desarrollo de una implementación del mecanismo de fragmentación para RADIUS. Se pretende realizar un buen diseño de la implementación para la obtención de un código modular y reutilizable, de forma que quede lo más sencillo y eficiente posible.

Por otro lado, otra parte del diseño se centra en examinar el *código del software* del que partiremos, para buscar la forma y lugar en el que implementar la solución de fragmentación para RADIUS. Se ha buscado añadir dicha funcionalidad sin entorpecer, modificar o eliminar alguna de las funciones que ya incorpora RADIUS.

4.1 Motivación del diseño

En primer lugar, vamos a explicar qué es la fragmentación en RADIUS, la motivación de la misma y qué nos permite conseguir con ella. La necesidad de fragmentación comienza cuando es necesario enviar información dentro de paquetes RADIUS que superen los 4096 bytes. Dicha cifra es el límite que impone el protocolo RADIUS para el tamaño de los paquetes. Pero, ¿por qué la realización e implementación de esta propuesta?

Cuando apareció la necesidad de enviar datos grandes en RADIUS, surgieron varias posibles soluciones. Por ejemplo, el RFC 6158 [28] plantea un esquema para tratar con datos grandes. En dicho RFC se establecía el intercambio de una secuencia de *Access Challenge* para enviar esos datos, pero no es válida porque muchos atributos RADIUS que contienen datos que necesitan fragmentación no pueden ser transportados en este tipo de paquetes. También se establecía otra solución, enviar en estos paquetes nombres en vez de valores, es decir, “dato1” puede significar “conceder acceso al puerto 80”. De esta forma, en los paquetes *Access Challenge* iría “dato1”, y cuando el otro extremo lo recibe y lee esa información, sabe que realmente se refiere a “conceder acceso al puerto 80”. Como problema presenta que sólo es válido para datos estáticos (no generados de forma dinámica). Esto impide el transporte de información de autorización generada de forma dinámica. Por ejemplo, SAML se genera de firma dinámica haciendo que no sea válida esta solución.

Otra solución, quizás la más sencilla, sería eliminar la limitación de 4096 bytes en el propio estándar. Realmente el campo *longitud* del paquete RADIUS tiene 64 kb. Pero, no es una solución factible, ya que eliminar dicha restricción implicaría un cambio en el protocolo RADIUS, y por lo tanto, requiere actualizar todas las entidades desplegadas actualmente como proxies, servidores, etc.

Por estos motivos, se decide crear un mecanismo de fragmentación que permita el envío de grandes cantidades de datos de autorización (por encima de 4096 bytes) a través del protocolo de transporte RADIUS. Se centra en datos de autorización, ya que por ejemplo, la autenticación se basará en algún método EAP y éstos ya definen su propio mecanismo de fragmentación. Como detalle importante, el mecanismo de fragmentación no implica la modificación de las infraestructuras existentes. Sólo es necesario actualizar los nodos implicados en la fragmentación.

4.2 Descripción del proceso de fragmentación

4.2.1 Concepto de chunk y atributos de fragmentación

Este proceso de fragmentación define una serie de conceptos nuevos que deben ser explicados.

En primer lugar, aparece el concepto de *chunk*. Un paquete grande que excede de tamaño se divide en *chunks*. Estos *chunks* son paquetes RADIUS estándar, y por tanto, perfectamente válidos para cualquier infraestructura RADIUS existente. Son los encargados de enviar los datos fragmentados. Tienen las siguientes características.

- Tienen el mismo tipo que el paquete original a fragmentar.
- Transportan un subconjunto de los atributos. Es decir, el contenido de cada *chunk* será un subconjunto de atributos del paquete original.
- Irán señalizados mediante dos atributos RADIUS (a excepción del último *chunk*) indicando el estado de la fragmentación.

De este modo, a partir de estos *chunks*, se podrá reconstruir el paquete RADIUS original que excedía de tamaño.

Por otro lado, en la propuesta se incluye la definición de nuevos atributos RADIUS. Serán utilizados por los *chunks* para señalizar el proceso de fragmentación. A su vez, se utilizarán atributos ya existentes como apoyo en el proceso de fragmentación. Los dos atributos que se definen en la propuesta para el proceso de fragmentación son:

- *Frag-Status*. Este atributo se utiliza para conocer el estado del proceso de fragmentación. Puede tener varios valores, e indicará si quedan datos por enviar, si se están solicitando más, o si ha acabado el proceso. Debe aparecer siempre que exista un intercambio de fragmentación salvo en el último *chunk* enviado, ya que realmente no es un paquete fragmentado. Los valores son:
  - 0: valor reservado.
  - 1: indica que el cliente o servidor soporta fragmentación.
  - 2: indica que quedan datos pendientes por enviar (More-Data-Pending).
  - 3: indica a la parte que inició la fragmentación que siga enviando datos (More-Data-Request).
- *Proxy-State-Len*. Este atributo es utilizado para indicar cuantos bytes deben ser reservados a la hora de enviar un *chunk* para que los proxies intermedios incluyan el atributo *Proxy-State*. Sólo se utiliza en la fragmentación realizada en el flujo de pre-autorización, ya que este valor sólo es necesario que lo conozca el cliente a la hora de calcular el tamaño útil del chunk (tamaño que queda libre en el paquete para añadir información). Se debe a que él no puede saber cuantos atributos Proxy-State se incluirán en el camino.

Por otro lado, a continuación aparecen los atributos ya existentes en RADIUS que se utilizarán como apoyo en el proceso de fragmentación.

- *Service-Type*. Se añade a todos los *chunks,* y a sus respuestas, para indicar que un paquete *Access Accept* no garantiza aún el acceso al servicio. Esto viene motivado porque aún se debe terminar el intercambio de información adicional de autorización. Por esta razón, el valor que tendrá este atributo en todos los *chunks* es Additional-Authorization (200). Ese valor es nuevo y se añade a este atributo ya existente para el proceso de fragmentación.

- *State*: se utiliza en el proceso de fragmentación RADIUS para vincular todos los *chunks* que pertenecen a un mismo paquete fragmentado. De esta forma, permite al servidor conocer a qué proceso de fragmentación pertenece una respuesta a un chunk.

- *Proxy-State*. Este atributo es añadido por los proxies intermedios entre los dos extremos. Requiere de un tratamiento especial en el proceso de fragmentación, ya que el servidor RADIUS deberá copiar todos los *Proxy-State* en el paquete de respuesta. A su vez, calculará el tamaño que ocupan todos los atributos *Proxy-State* que contenga el paquete, para notificar al cliente mediante el uso del atributo *Proxy-State-Len*.

- *User-Name*. Este atributo es utilizado por proxies para cuestiones de enrutamiento de los paquetes. Es por ello, que la propuesta indica que este atributo debe aparecer en todos los paquetes enviados desde el cliente hasta el servidor RADIUS.

- *Message-Authenticator*. Debe aparecer en todos los *chunks* enviados en el proceso de fragmentación, así como en las respuestas a los mismos. Su uso proporciona protección y seguridad a todos los paquetes enviados durante el proceso de fragmentación.

Esta propuesta indica que en un proceso de fragmentación iniciado por el servidor, el cliente nunca deberá almacenar los atributos *Frag-Status* y *Service-Type* (siempre que su valor sea *Additional-Authorization* o aparezca en el último *chunk*). Tampoco deberá almacenar el atributo *Proxy-State-Len* y el atributo *State*, excepto el del último *chunk* si está presente ya que se corresponde con el del paquete original.

Si el proceso de fragmentación fue iniciado por el cliente, el servidor no almacenará los atributos *Frag-Status*, *Service-Type*, (si su valor es *Additional-Authorization*) ni *Proxy-State* (excepto los que aparezcan en el último *chunk)*. Tampoco podrá almacenar los atributos *User-Name* y *State*, salvo los presentes en el primer *chunk* recibido.

Por otro lado, el servidor deberá enviar en el último *chunk* los atributos *State* y *Service-Type* del paquete original, si existen. Por otra parte, el cliente deberá enviar en el primer *chunk* los atributos *State* y *User-Name* del paquete original, si existen.

4.2.2 Fases de la fragmentación

En este apartado vamos a ver las fases del proceso de fragmentación y los detalles de cada una. La propuesta indica que deben existir 3 fases. Una primera llamada flujo de *pre-autorización*, luego *autenticación* y por último *post-autorización*.

- Pre-autorización. Este flujo lo inicia el cliente RADIUS y se envía hacia el servidor RADIUS. Es utilizado para enviar información de autorización que

puede ser necesaria a la hora de realizar la autenticación por parte del servidor RADIUS. Si la información a enviar supera los 4096 bytes se llevará a cabo el proceso de fragmentación.

- Autenticación. En este flujo se lleva a cabo la autenticación del cliente por parte del servidor RADIUS. Es iniciado por el NAS y durante el intercambio se transmite la información necesaria para llevar a cabo la autenticación del cliente en función del método de autenticación elegido. En esta fase la propuesta de fragmentación no se aplica debido a que los métodos EAP utilizados para autenticar ya definen su propio mecanismo.

- Post-autorización. Este flujo se inicia por parte del servidor RADIUS hacia el NAS. En él se envía la respuesta generada después del proceso de autenticación. En dicha respuesta puede enviarse información de autorización generada después de la autenticación para que el cliente o el propio NAS la procese. Si dicha información supera los 4096 bytes se inicia el proceso de fragmentación.

La **Figura 10** resume cuándo tiene lugar cada flujo de mensajes.

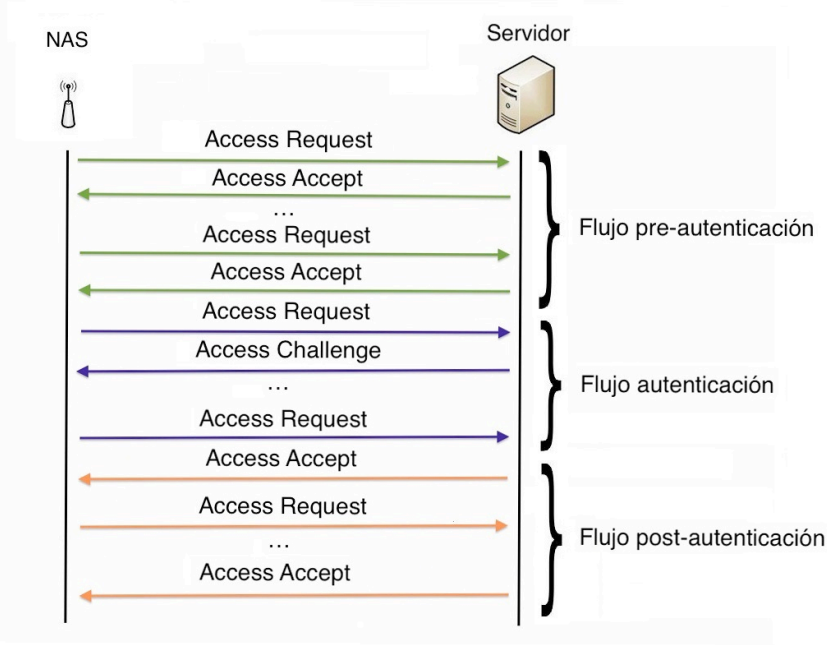

Figura 10: Flujo de mensaje RADIUS con fragmentación

En general, la propuesta presenta una solución para el envío de cualquier tipo de información que requiera fragmentación, tanto en las fases de pre-autorización como de post-autorización. Sin embargo, en este trabajo, se va a utilizar SAML como lenguaje para representar dicha información de autorización. Esta decisión se debe a que SAML, como se ha visto en la sección 2, es la solución elegida por ABFAB/Moonshot, y ya se ha comentado que uno de los objetivos de este trabajo es la integración de esta implementación con la propuesta de ABFAB/Moonshot. Otras alternativas a SAML podrían ser estándares como JSON [42], CPL (CiscoPolicyLanguage) [47], etc.

4.2.2.1 Flujo de pre-autorización

Supongamos que queremos enviar un *SAMLAuthenticationRequest* con información a utilizar en el proceso de autenticación. Estos datos, si superan el tamaño máximo, se dividirán en *chunks* (mensajes *Access Request* válidos) y se enviarán al servidor a través de un intercambio de múltiples *Access Request / Access Accept*. A continuación, *en la **Figura 11**, podemos* ver un ejemplo de un *Access Request* con un *SAMLAuthenticationRequest* que deberá ser fragmentado.

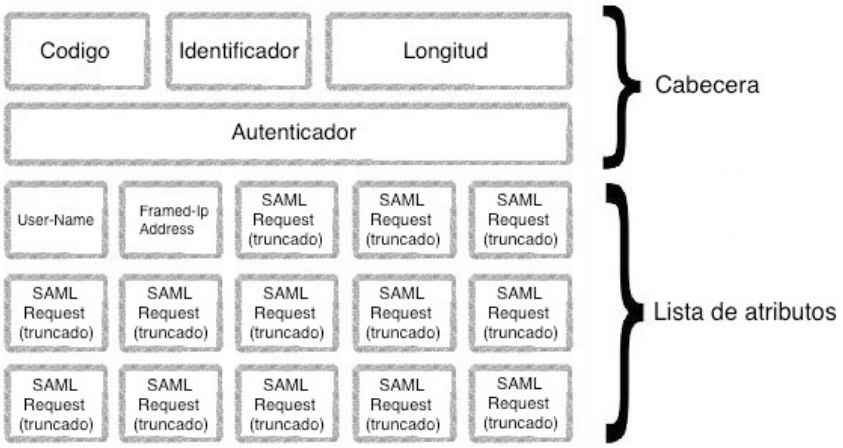

Figura 11: Contenido del paquete Access Request original

Aunque en la sección 2 ya comentamos que el tamaño máximo para un paquete RADIUS es de 4096 bytes y el tamaño máximo de un atributo son 255 bytes, para este ejemplo, imaginemos que el límite de paquete es de 8 atributos y que el SAML*AuthenticationRequest* del paquete será dividido en 13 atributos que contendrán la información (atributos *SAMLRequest*). Por lo tanto el paquete *Access Request* original deberá ser fragmentado en tres *chunks*. En la **Figura 12** podemos ver el intercambio llevado a cabo, junto con los atributos que irán en cada paquete RADIUS y su valor en algunos.

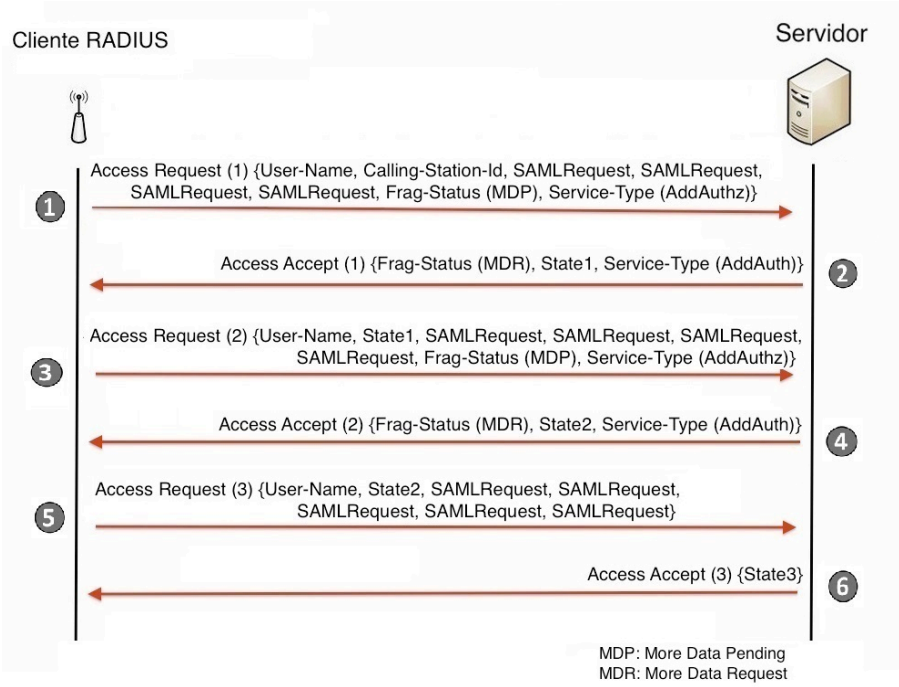

Figura 12: Flujo de mensajes en pre-autorización

A continuación vamos a explicar cómo se ha llevado a cabo el flujo y los detalles de cada mensaje.

1. Un usuario solicita acceso a un recurso y el cliente RADIUS genera el paquete original. Al existir fragmentación, el cliente RADIUS genera el primer *chunk*. Se trata de un *Access Request* al que añade cuatro atributos con parte del contenido *SAMLRequest*. También se envía el atributo *User-Name* y *Calling-Station-Id* del paquete original (incluido el atributo *State* si también existe en el paquete original), y los atributos de fragmentación como, *Frag-Status* con valor More-Data-Pending y *Service-Type* con valor Additional-Authotization. El primer chunk queda completo, y no podría añadirse ningún atributo más.

2. En este punto, el servidor recibe el *chunk*. Lo procesa y almacena los atributos correspondientes recibidos en el chunk que formarán parte del paquete original, incluidos los que contienen la información relativa al *SAML Request* (atributos *SAMLRequest*). Construye la respuesta en un *Access Accept* para enviársela al cliente y que éste siga enviándole la información restante. Dicho paquete contiene los atributos de fragmentación *Frag-Status* con valor More-Data-Request y el atributo *Service-Type* con valor Additional-Authorization, tal y como indica la propuesta. También incluye el atributo *State*, con el fin de vincular las solicitudes posteriores que lleguen. Por último, aparecerá el atributo *Proxy-State-Len* que indicará los bytes que deben ser reservados por el cliente en este proceso flujo para que los proxies (si existen) añadan los atributos *Proxy-State*.

3. El cliente recibe el paquete *Access Accept* del servidor y prosigue con el proceso de fragmentación. De nuevo construye un *Access Request* similar al anterior, con cuatro atributos relativos al *SAML Request* y los dos relacionados con la fragmentación (*Frag-Status* y *Service-Type*). Añade una copia del atributo *State* recibido del servidor.

4. De la misma manera, el servidor recibe este segundo *chunk*, almacena los nuevos atributos con los anteriores y vuelve a enviar una respuesta al servidor exactamente igual a la anterior. Simplemente variará el valor del atributo *State*, poniendo un valor nuevo para asociar el paquete posterior.

5. El cliente envía el último *chunk*. Este paquete *Access Request* no contendrá ningún atributo de fragmentación, ya que dicho proceso ha terminado. En él aparecen el resto de atributos que contienen el *SAML Request* (atributos *SAMLRequest*), el atributo *User-Name* y una copia del atributo *State* enviado en el mensaje anterior. En este punto, el cliente ya ha enviado toda la información de pre-autorización.

6. El servidor recibe el último *chunk* (lo reconoce ya que no contiene atributos de señalización de fragmentación como Frag-Status y Service-Type). En este punto el servidor concatena internamente todos los atributos, tanto los recibidos anteriormente como los de este último *chunk*. Reconstruye el paquete original y lo procesa. Responde al cliente indicando que ha acabado correctamente el flujo de pre-autorización.

Los atributos *Message-Authenticator* los obviamos de la **Figura 12** y de la explicación ya que van en todos los paquetes del intercambio. Los utilizamos en el intercambio para dotar al *chunk* de seguridad.

Cuando el cliente reciba el último *Access Accept*, podrá continuar con el proceso de autenticación, pasando a la siguiente etapa.

4.2.2.2 Flujo de post-autorización

En este apartado vamos a detallar los aspectos relativos al flujo de post-autorización descrito anteriormente. Dicho intercambio se producirá mediante intercambios *Access Accept / Access Request* y lo iniciará el servidor después del proceso de autenticación con EAP. En nuestro caso, enviaremos un *SAMLResponse* que contiene uno o varios *Assertions SAML*. Estos datos, si superan el tamaño máximo, se dividirán en *chunks* y se enviarán al cliente. A continuación, en la **Figura 13**, podemos ver un ejemplo de un *Access Accept* con un *SAML Response* que deberá ser fragmentado.

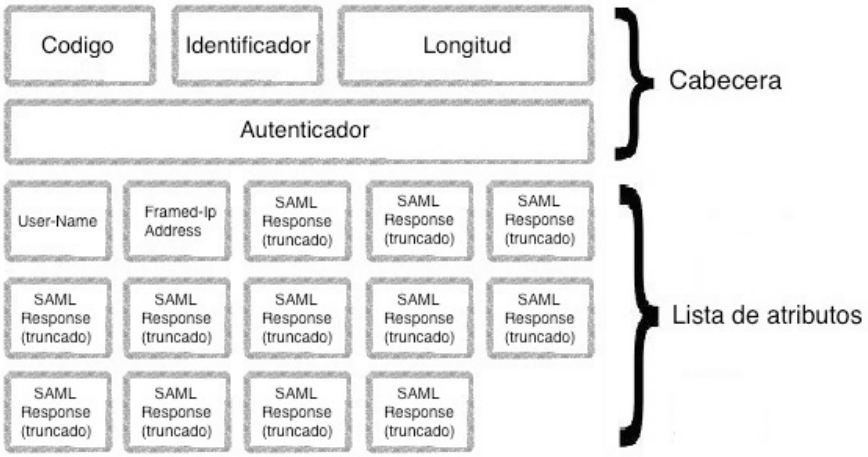

Figura 13: Contenido del paquete Access Accept original

Del mismo modo que en el flujo anterior, imaginemos que el *SAMLResponse* del paquete es dividido en 12 atributos que contendrán la información (atributos *SAMLResponse*), y el *Access Accept* deberá ser fragmentado en tres *chunks* debido a que el límite de paquete sigue siendo 8 atributos. En la **Figura 14** podemos ver el intercambio llevado a cabo, junto con los atributos que irán en cada paquete RADIUS y su valor.

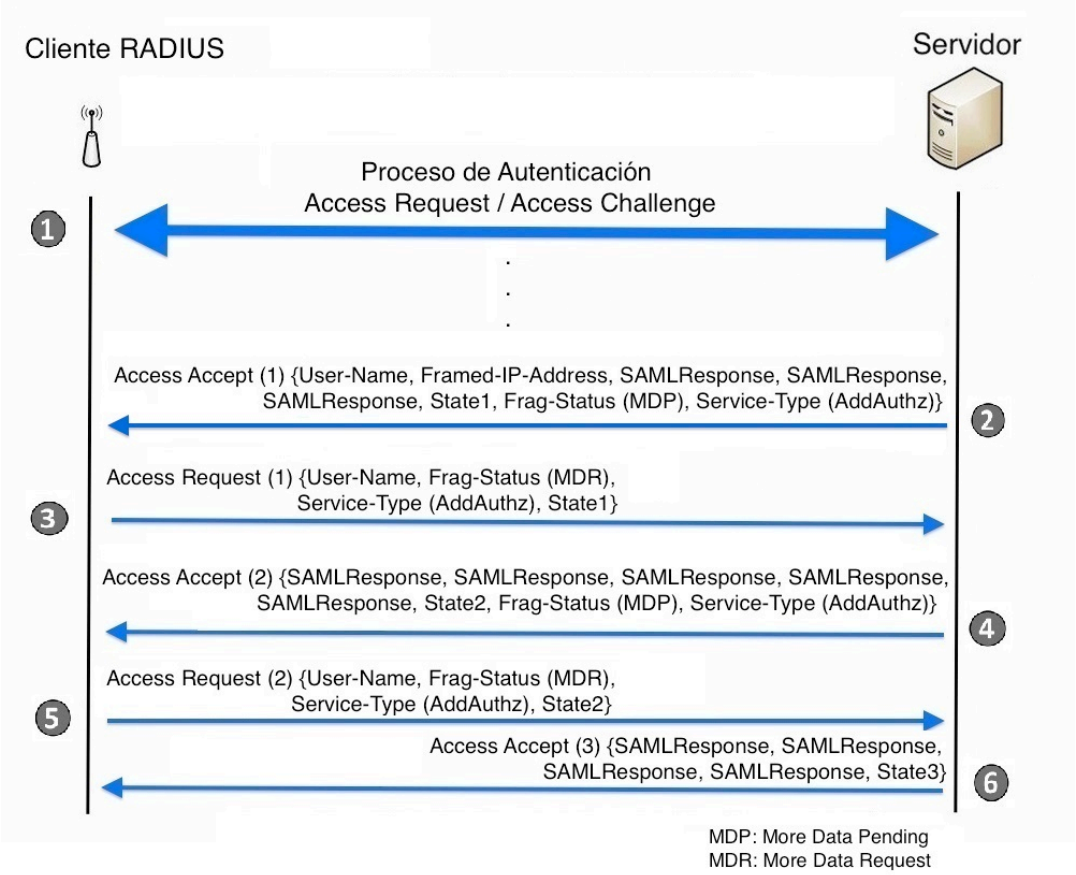

Figura 14: Flujo de mensajes en post-autorización

A continuación vamos a explicar cómo se ha llevado a cabo el flujo y los detalles en cada punto.

1. Se realiza el proceso de autenticación. Si todo es correcto, el servidor generará un mensaje *Access Accept* con los atributos (SAMLResponse) que contienen el *SAMLResponse* hacia el cliente.

2. El mensaje creado en el punto 1 debe ser fragmentado. Se crea el primer *chunk* (mensaje *Access Accept*) que contiene tres atributos con el contenido del *SAMLResponse* (atributos *SAMLResponse*). Se añade el atributo *User-Name* y *Framed-IP-Address* del paquete original, y los atributos de fragmentación como, *Frag-Status* con valor More-Data-Pending y *Service-Type* con valor Additional-Authorization. Por último se añade el atributo *State* generado durante el proceso de fragmentación que permitirá al servidor vincular los paquetes posteriores con este proceso de fragmentación.

3. El cliente recibe el chunk, lo procesa almacenando los atributos recibidos que deben ser almacenados y construye una respuesta. Irá en un paquete *Access Request* con los atributos de fragmentación (*Frag-Status* con valor More-Data-Request y *Service-Type* con valor Additional-Authorization). También añade el atributo *User-Name* y una copia del atributo *State*.

4. El servidor recibe el paquete desde el cliente y prosigue con el proceso de fragmentación. Construye un nuevo *chunk* que contendrá de nuevo todos los atributos que puedan ser añadidos con el contenido del *SAMLResponse* (atributos *SAMLResponse*). Añade de nuevo los dos atributos de fragmentación indicando que el proceso continua y un nuevo atributo *State* para seguir vinculando los paquetes posteriores.

5. De la misma manera, el cliente lo recibe, almacena los atributos nuevos que deben ser almacenados junto a los que llegaron posteriormente y construye la respuesta. Irá en otro mensaje *Access Request* y contendrá los atributos de fragmentación (Frag-Status y Service-Type) solicitando más *chunks*, el atributo *User-Name* y la copia del atributo *State* que se acaba de recibir.

6. Nuevamente el servidor recibe el paquete y enviará un nuevo *chunk*. En este caso, el proceso de fragmentación se da por concluido ya que no hace falta fragmentar los datos restantes y se procede a enviar el último *chunk* hacia el cliente. Este paquete llevará el resto de atributos que quedan del *SAMLResponse* (*SAMLResponse*) y un atributo *State* y *Service-Type* si existían en el paquete original. No se incluyen los atributos de fragmentación ya que el proceso *se ha dado por concluido*.

Cuando el cliente reciba este último *Access Accept se dará por concluido* el proceso de fragmentación. El cliente recompondrá el paquete original que contiene el *SAMLResponse* utilizando todos los *chunks* recibidos, finalizando el flujo de post-autorización.

4.2.3 Discusión de los aspectos de diseño

El mecanismo de fragmentación ha provocado ciertos aspectos de discusión dentro del grupo de trabajo que se encargó de diseñarlo (IETF RADEXT WG [48]). Estos aspectos derivaron en ciertas consideraciones que deben tenerse en cuenta a la hora de implementarlo.

- **Operación con proxies RADIUS.** La propuesta define dos modos de operación para los posibles proxies que puedan existir en el camino durante un proceso de fragmentación.

  - *Proxy "Legacy"*. La propuesta indica que la fragmentación debe ser transparente a estos equipos. Ellos verán el chunk como un paquete normal y lo encaminaran en base al atributo *User-Name*. No podrán añadir, borrar o modificar los atributos del paquete original, sólo podrán añadir el atributo *Proxy-State*.

  - *Proxy "Updated".* La propuesta también indica que pueden existir proxies que sean conscientes de la fragmentación. En este caso, los proxies pueden modificar el paquete original, añadiendo, eliminando atributos, etc. De esta manera, *los proxies updated* pueden obtener todos los *chunks* del paquete original, recomponerlo y añadir o

modificar atributos para encaminarlo hacia el servidor posteriormente.

- **Tamaño útil del *chunk*.** El tamaño útil del chunk es el tamaño que quedará libre en el paquete para añadir la información de pre-autorización y de post-autorización una vez añadidos los atributos requeridos por el proceso de fragmentación. Este tamaño útil se ve afectado por los atributos de fragmentación. Por ejemplo, todos los *chunks* deben llevar el atributo *Frag-Status* y *Service-Type* obligatoriamente. Por otro lado, pueden existir proxies que incluyan atributos *Proxy-State* afectando al tamaño útil del *chunk*. De esta forma, el tamaño que se podrá utilizar del *chunk* variará en función de estos factores.

- **Consideraciones de seguridad.** El mecanismo de fragmentación ni añade ni cambia nada del mecanismo de seguridad de RADIUS. Es por ello, que la propuesta debe aplicar las consideraciones de seguridad establecidas por el propio RADIUS. Por ejemplo, se requiere que todos los *chunks* lleven el atributo *Message-Authenticator* para que vayan protegidos. A su vez, también se recomienda en la propuesta limitar el tamaño máximo de datos a recibir. Siendo un **máximo** de **64 KBytes** o no más de **20 *chunks***.

- **Violación del RFC 2865 [2].** La propuesta indica que el flujo de autenticación queda pospuesto hasta que haya finalizado el flujo de pre-autorización y se haya podido recomponer el paquete original. Esto provoca, que cuando RADIUS reciba el primer paquete *Access Request* original puede que no contenga alguno de los tipos de atributos de autenticación esperados (i.e. EAP-Start) tal y como requiere el RFC 2865. Esto provocaría un fallo de autenticación. Por lo tanto, se propone como solución dar como válidos los paquetes de fragmentación pertenecientes al flujo de pre-autorización, solución que se ha consensuado dentro del grupo de trabajo RADEXT del IETF.

- **Uso de SAML y atributo *SAML-AAA-Assertion*.** Por último, aunque la propuesta presenta una solución genérica, para nuestra implementación hemos decidido utilizar SAML para transportar la información de pre-autorización y de post-autorización. Para ello, un aspecto del diseño de nuestra implementación es decidir en que tipo de atributo viajará la información SAML entre los extremos durante la fragmentación. Para ello decidimos utilizar el atributo *SAML-AAA-Assertion* definido en el diccionario *Ukerna* [43]. Esta decisión en parte fue motivada por el uso de ABFAB/Moonshot, ya que dicho software incorpora este atributo para el envío de información SAML.

4.3 Implementación de la solución

En esta sección vamos a explicar detalles de la implementación de fragmentación en el servidor y cliente RADIUS, para una versión standalone (versión independiente del software), y para su integración con ABFAB/Moonshot.

4.3.1 Software utilizado

4.3.1.1 FreeRADIUS

FreeRADIUS versión 2 [11] es una distribución software de código abierto que incluye un servidor RADIUS, y unos clientes RADIUS simples, radclient y radeapclient, que usaremos en nuestra implementación. Actualmente se encuentra en la versión 3 que incorpora soporte para RADIUS sobre TLS incluyendo RadSec. No obstante en

nuestro caso hemos utilizado la versión 2, pues es la denominada estable y nos va a proporcionar la utilidad que necesitamos.

A su vez, hemos empleado dos clientes RADIUS que incorpora FreeRADIUS para realizar un cliente básico con soporte de fragmentación que pruebe la funcionalidad y diseño llevados a cabo. Posteriormente hemos integrado la fragmentación en ABFAB/Moonshot, permitiendo llevar la propuesta a un escenario realista.

La realización de la implementación de ambas partes se ha llevado a cabo en el lenguaje C [45], ya que así está programado el software que vamos a utilizar.

4.3.2 Implementación de servidor

En este apartado vamos a explicar con todo detalle los pasos que se han llevado a cabo para implementar en el servidor toda la funcionalidad de fragmentación.

4.3.2.1 Estructuras de datos para la fragmentación

El siguiente paso es crear el diseño de las estructuras de datos. Hemos decidido crear una estructura de datos en el servidor que represente un estado de fragmentación. Será utilizada para almacenar los datos pendientes por enviar en el flujo de post-autorización o los que se van recibiendo durante el flujo de pre-autorización. A continuación la detallamos.

Su nombre es *struct server_frag_state_t*. Esta estructura de datos es usada por el servidor para representar un estado de fragmentación. Dispone de los siguientes campos:

```
typedef struct server_frag_state_t {
    int initiator;
    VALUE_PAIR* state;
    VALUE_PAIR* list;
    struct server_frag_state_t *next;
} server_frag_state_t;
```

Observamos los campos *state* y *list*, ambos del tipo *VALUE_PAIR*. Este tipo representa un par atributo-valor explicado en el punto 2.2.2 de este trabajo. El primero será utilizado para vincular un paquete recibido a un proceso de fragmentación en curso. El segundo (campo *list*) contendrá los atributos del paquete original pendientes de enviar en post-autorización o los recibidos en pre-autorización. A su vez, el campo *initiator* indica quién inició la fragmentación, indicando con un 1 que la inició el servidor, o con un 0 que fue el cliente. De esta forma sabremos si el estado de fragmentación se corresponde al flujo de pre-autorización o de post-autorización.

A su vez, existirá una lista en el servidor que contendrá todos los estados de fragmentación que haya en curso. Puede darse el caso, en el que reciba varias peticiones de forma concurrente y tenga a la vez varios procesos de fragmentación en curso.

Por otro lado, en el servidor existirá una constante que nos indicará el tamaño máximo que podrá tener un paquete. En base a ella haremos los cálculos necesarios para saber si hay que fragmentar o no.

```
#define MAX_CHUNK_LEN   4096
```

4.3.2.2 Funciones auxiliares

Nos encontramos una función que se encargará de comprobar si es necesaria o no la fragmentación de un paquete. Se llama *fragmentation_required* y su valor de retorno es un entero que actuará como booleano. Su cabecera es:

```
static int fragmentation_required(RADIUS_PACKET *packet, const RADIUS_PACKET
*original, const char *secret, int max_length, VALUE_PAIR**
out_last_attribute)
{
…
}
```

Como parámetros encontramos el paquete RADIUS del que queremos saber su tamaño, otro campo llamado original (necesario para llamar a la función *rad_vp2att* que veremos después), el secreto (es la clave con la que se cifran los atributos que deben ir cifrados) y el tamaño que nos queda libre en el paquete (*MAX_CHUNK_LEN* menos el espacio de los atributos que deben ir fijos. Por ejemplo: *Frag-Status*, *Message-Authenticator, etc.*). Por último, hay un parámetro que es usado como retorno para devolver el atributo RADIUS que indica cual es el último atributo que se incluirá en el *chunk* en el caso en el que haya que fragmentar. De ese modo, desde el atributo número uno hasta el devuelto irían en el primer chunk, el resto los debemos almacenar para intentar enviarlos en mensajes posteriores.

La forma de conocer si es necesaria la fragmentación se basa en calcular uno a uno el tamaño codificado de cada atributo e ir sumando el valor. Si en algún momento sobrepasa el parámetro *max_length* indicará que es necesario fragmentar.

La estimación del tamaño de los atributos cuando son codificados se realiza mediante la función *rad_vp2att* que incorpora FreeRADIUS. Dado un atributo lo codifica para enviarlo por la red y devuelve el tamaño que ocupa.

4.3.2.3 Funciones principales

4.3.2.3.1 Pre-autorización

En primer lugar vamos a explicar la implementación llevada a cabo para la función que se encarga de gestionar la pre-autorización en el servidor. El nombre escogido es *server_process_fragmentation_request*. La cabecera de la función es:

```
int server_process_fragmentation_request(RADIUS_PACKET *request_packet,
RADIUS_PACKET *reply_packet)
{
…
}
```

Recibe dos parámetros, *request_packet* que se corresponde con el paquete recibido por la red sobre el que vamos a trabajar, y *reply_packet*, que es el paquete en el que crearemos la repuesta.

En primer lugar extraemos los atributos relacionados con la fragmentación (*Frag-Status, State y Service-Type*) del paquete recibido y se comprobará si existe un estado de fragmentación en base al atributo *State*.

El siguiente paso es tratar cada uno de los diferentes casos que podemos tener.

1. Uno de ellos es que el paquete no se corresponda con un proceso de fragmentación. Ocurrirá si no existen atributos de fragmentación en el paquete ni un estado de fragmentación anterior alojado en el servidor.

2. Otro de los posibles casos es que el paquete se corresponda con el último *chunk* de un proceso de fragmentación. Para ello no llevará ni el atributo *Frag-Status* ni *Service-Type*, pero si existirá en el servidor un estado de fragmentación. En este caso concatenaremos los atributos que contiene el paquete recibido junto con los que se encontraban en el servidor almacenados (recomponemos el paquete original). Previamente hemos eliminado los atributos que no deben ser almacenados (visto en la sección 4.2.1). Una vez hecho esto construimos la respuesta para el otro extremo. Añadiremos un atributo *State* y el atributo *Message-Authenticator*, tal como vimos en la sección 4.2.2.1. En este caso, el valor de retorno será 2. Indicará al servidor que el mensaje que se ha recibido (el mensaje original reconstruido a partir de todos los *chunks*) se corresponde con un proceso de pre-autenticación y por lo tanto se dará como válido el flujo y el mensaje (este caso fue analizado en las discusiones del punto 4.2.3).

3. Por último, queda comprobar si el paquete recibido contiene el atributo *Frag-Status* con valor More-Data-Pending y también el atributo *Service-Type* con valor Additional-Authorization. Eso nos indica que el paquete recibido es un *chunk*, y por lo tanto contiene atributos que debemos almacenar. Lo primero que haremos es extraer del paquete los *Proxy-State* y estimar el tamaño que ocupan para indicarle en el atributo *Proxy-State-Len* este valor al otro extremo. El siguiente paso es comprobar si existe un estado de fragmentación en base al atributo *State* del paquete. Podemos encontrarnos con dos situaciones:

   - Que no exista un estado de fragmentación anterior. Si no existe, se trata del primer chunk recibido, es el comienzo del flujo de pre-autorización. Para ello, hay que crear un estado de fragmentación que contenga los atributos que hemos recibido en el chunk, excepto *Frag-Status* y *Service-Type*. En este *chunk*, al ser el primero, se almacenarán los atributos *State* y *User-Name* si existen.

   - Que exista un estado de fragmentación anterior. En este caso, tenemos que añadir los atributos recibidos con los que se encuentran almacenados en el estado de fragmentación. Deberemos eliminar en este caso, tanto los atributos de fragmentación como los atributos *State* y *User-Name*. A su vez, se actualizará el atributo *State* almacenado en el estado de fragmentación, para que a la hora de crear la respuesta se envíe este nuevo valor y poder asociar la respuesta que recibamos después.

   Una vez almacenados los atributos se construye la respuesta al cliente. Este paso es común para las dos situaciones que acabamos de ver ahora mismo. Añadiremos los atributos *Frag-Status* con valor More-Data-Request, *Service-Type* con valor Additional-Authorization y una copia del atributo *State* que hay almacenado en el estado de fragmentación. También añadiremos el atributo *Proxy-State-Len*. El valor de este atributo es el que estimamos anteriormente utilizando los atributos *Proxy-State* que contenía el *chunk*. Posteriormente, reintroduciremos los atributos *Proxy-State* si es que los había en el *chunk*, y si no existiera en el paquete, añadimos el atributo *Message-Authenticator*.

Los valores de retorno de la función serán varios. Como vimos antes, *devolverá 2* si es el último *chunk*, *1* si hemos recibido un *chunk*, ya sea primero o uno intermedio, y 0 en el caso en el que no se corresponda con un paquete de fragmentación o exista algún error.

4.3.2.3.2 Post-autorización

El siguiente paso es explicar las dos funciones que se encargan de gestionar el flujo de post-autorización. En primer lugar vamos a explicar la función *server_create_fragmentation_request*. Se encarga de gestionar la fragmentación en el flujo de post-autorización, fragmentando un paquete Access Accept si fuera necesario. El esqueleto de la función es:

```
int server_create_fragmentation_request(RADIUS_PACKET *packet, const
RADIUS_PACKET *original, const char *secret)
{
…
}
```

Tiene como parámetros el paquete RADIUS a enviar, otro parámetro (*original*) que es un paquete RADIUS y el secreto, que como acabamos de ver, ambos son necesarios para la función *fragmentation_required*.

En primer lugar habrá que extraer los atributos *State* y *Service-Type* para añadirlos al final de la lista de atributos, ya que si existen son los que contiene el paquete original y deben enviarse en el último *chunk* tal y como vimos en sección 4.2.1. Los atributos *Proxy-State* del paquete original, si existen, se utilizarán para realizar la estimación de los bytes que hay que reservar en el nuevo *chunk* para incluir estos atributos.

El siguiente paso es calcular el tamaño útil del *chunk*, restando los bytes que ocupan los atributos que deben ir fijos.

```
chunksize = MAX_CHUNK_LEN - proxystatelen - 60;
```

La variable *chunksize* es el tamaño útil que tendrá el chunk, es decir, cuantos bytes de datos se podrán enviar. Se calcula restando el tamaño total que puede tener el *chunk* a la estimación de los atributos *Proxy-States* y a su vez, restándole 60 bytes. Este valor 60 viene de calcular el tamaño de los atributos de señalización de fragmentación.

El siguiente paso es comprobar si todos los atributos del paquete original pueden ser enviados sin superar el tamaño útil del paquete (variable *chunksize*). La función *fragmentation_required* nos dirá si es necesario realizar dicha fragmentación o no en el paquete. Es muy importante resaltar que esta función recibe como parámetro la variable *chunksize*.

En función del resultado, podría ser necesario fragmentar o no. Si no es necesario (porque el paquete original es pequeño o es el último *chunk*) se añadirán los atributos *Message-Authenticator* y *State* si no existen en el paquete. Si por el contrario es necesario fragmentar, se debe crear el *chunk* y almacenar el resto de atributos que no pueden ser enviados. Para ello se crea el estado de fragmentación del servidor (*server_frag_state*) que almacenará los atributos que no han podido ser enviados.

El siguiente paso es añadir al *chunk* todos los atributos referentes al proceso de fragmentación. Añadimos el atributo, *Frag-Status*, con valor More-Data-Pending y *Service-Type* con valor Additional-Authorization. También se añade el atributo *State*

con un valor creado aleatoriamente para el proceso de fragmentación y que como ya vimos, permitirá vincular con este proceso en curso los mensajes recibidos posteriormente.

Finalmente, se añaden los atributos *Proxy-States* extraídos al principio para calcular la estimación del tamaño que ocupan, y se comprueba si existe el atributo *Message-Authenticator* para añadirlo o no. Con todo esto, el paquete estaría preparado para enviarlo por la red, sea un *chunk* o un paquete original.

La siguiente función se encarga de comprobar si el paquete recibido por el servidor se corresponde con la respuesta a un *chunk* enviado previamente en el flujo de post-autorización. De ser así debemos seguir enviando los atributos que quedan pendientes. Su nombre es *server_process_fragmentation_response* y a continuación vamos a explicar con más detalle el proceso que se lleva a cabo.

```
int server_process_fragmentation_response(RADIUS_PACKET *request,
RADIUS_PACKET *reply)
{
…
}
```

Esta función sólo recibe como parámetro dos paquetes RADIUS. Uno de ellos es el que contiene el mensaje recibido por la red (*request*) y el otro (*reply*) es el que utilizaremos para construir el siguiente chunk y comprobar si necesita ser fragmentado o no.

Inicialmente, buscaremos en el paquete recibido los atributos relativos a la fragmentación (*Frag-Status* y *Service-Type*) y también el atributo *State*. Con este atributo intentaremos recuperar un estado de fragmentación existente.

En base a la información que acabamos de recuperar, comprobaremos si es un paquete de fragmentación o no. Si no existe en el paquete recibido ningún atributo de fragmentación y tampoco hay un estado anterior de fragmentación alojado en el servidor, podemos asegurar que el paquete recibido no se corresponde con un proceso de fragmentación, y por lo tanto, no realizaremos ninguna acción y será procesado por RADIUS. Si por el contrario el paquete contiene el atributo *Frag-Status* con valor More-Data-Pending y el atributo *Service-Type* con valor Additional-Authorization, habremos recibido un *chunk* correspondiente al flujo de pre-autorización y será tratado en la función que vimos en la sección 4.3.2.3.1.

Por último, si el paquete contiene el atributo *Frag-Status* con valor More-Data-Request, el atributo *Service-Type* con valor Additional-Authorization y además hay un estado de fragmentación en el servidor podemos asegurar que el paquete es una respuesta a un *chunk* enviado anteriormente. Por lo tanto, tiene que continuar el proceso de fragmentación y enviar los atributos restantes.

Para ello, construiremos el siguiente *chunk*. Añadiremos al paquete todos los atributos que quedan pendientes en el estado de fragmentación (borrando dicho estado del servidor) y si el paquete no contiene el atributo *Message-Authenticator* se añadirá también.

Como valores de retorno, esta función devolverá un 1 (true) si el paquete recibido fue respuesta a un *chunk* enviado o 0 (falso) en cualquier otro caso (sea un *chunk*, un paquete normal o exista error).

La implementación y diseño de las funciones que darán soporte a los flujos de pre-autorización y de post-autorización queda finalizada con estas tres funciones principales.

4.3.2.4 Integración con FreeRADIUS

En esta sección, vamos a explicar cómo hemos integrado las funciones que acabamos de diseñar dentro del código de FreeRADIUS. También veremos cómo añadir los nuevos atributos y cómo generar los datos de post-autorización para añadirlos a un paquete. De esta forma, el servidor daría soporte a los flujos explicados anteriormente. Como vimos en la sección 4.2.1, tenemos que añadir nuevos atributos RADIUS y modificar los valores de algunos ya existentes. Para ello, dentro de la carpeta *share* de FreeRADIUS, encontramos distintos ficheros de diccionario. Unos almacenan los atributos básicos definidos en el RFC de RADIUS y otros tienen los atributos y valores de diccionarios añadidos para ser utilizados en distintos módulos de FreeRADIUS. En nuestro caso, en esta carpeta vamos a crear un fichero llamado *dictionary.tid*, que tendrá el siguiente contenido.

```
ATTRIBUTE       Large-Attr        21      string
ATTRIBUTE       Frag-Status       224     integer
ATTRIBUTE       Proxy-State-Len   220     integer
VALUE           Service-Type      Additional-Authorization    200
```

En primer lugar, añadimos el atributo *Large-Attr*, usado en las primeras versiones de esta implementación para enviar los datos adicionales de los dos flujos mencionados. Posteriormente pasará a quedar en desuso y se utilizará otro atributo ya existente en un diccionario[1] que incorpora FreeRADIUS (diccionario *Ukerna*). También añadimos el atributo *Frag-Status*, que aparecerá con identificador 224 y de tipo *integer*. El atributo *Proxy_State-Len* también es añadido, pero en este caso con identificador 220. Por último añadimos un nuevo valor al atributo ya existente *Service-Type*. Dicho valor es Additional-Authorization y tendrá como valor 200.

Las funciones auxiliares y principales, junto con las estructuras, las incluiremos en el fichero *radius.c.* En este fichero se encuentran las funciones que se encargan de gestionar los paquetes que se reciben y se envían por la red.

En primer lugar, vamos a ver dónde realizamos la llamada a la función *server_create_fragmentation_request*, definida en la sección 4.3.2.3.2. Hemos añadido dicha llamada en el fichero *listen.c*. En dicha clase se encuentran las funciones que se encargan de enviar un paquete por un socket hacia el otro extremo. En concreto, la llamada a nuestra función se encuentra en el método *auth_socket_send* y la podemos ver a continuación.

```
server_create_fragmentation_request(request->reply, request->packet,
                                    request->client->secret);
```

Por lo que, antes de enviar el paquete por el socket, llamaremos a nuestra función para comprobar si es necesario fragmentar el paquete para enviar siempre un paquete válido por la red, que no supere el límite de tamaño.

1 Inicialmente se utilizaba el atributo *Large-Attr* para el transporte de los datos de autorización. Cuando se diseño la integración en Moonshot se decidió utilizar el atributo *SAML-AAA-Assertion* para transportar esta información.

Por otro lado, en el fichero *event.c* tenemos los métodos que se encargan de manejar los eventos de entrada al servidor. Hay un método llamado *radius_handle_request* que se ejecuta cuando se recibe un paquete por la red. Por lo tanto, hemos añadido estas líneas al método justo después de recibir el paquete:

```
fragmentation = server_process_fragmentation_response(request->packet,
                                                      request->reply);
   if (!fragmentation) {
       fragmentation = server_process_fragmentation_request(request->packet,
                                                            request->reply);
       if(fragmentation == 2 ) {
          pairadd(&request->config_items, pairmake("Auth-Type","Accept",
                                                   T_OP_EQ));
          fragmentation = 0;
        }
   }
   if (!fragmentation)
         fun(request);
```

Con estas líneas podemos averiguar el tipo de paquete que hemos recibido antes de que lo trate FreeRADIUS, y en caso de ser un paquete relacionado con un proceso de fragmentación, hacer las operaciones oportunas que ya hemos comentado.

En primer lugar llamamos a la función *server_process_fragmentation_response* para saber si el paquete recibido es una respuesta a un *chunk* enviado por el servidor anteriormente (flujo de post-autorización). Si lo es, enviaremos la respuesta creada en dicha función. Por el contrario, si no lo es, llamaremos a la siguiente función, *server_process_fragmentation_request*. En ella comprobamos si el paquete recibido es un *chunk* (flujo de pre-autorización). Si no lo es, se trata de un paquete normal sin fragmentación, lo procesará FreeRADIUS. Si por el contrario es un chunk, de nuevo evitaremos el procesamiento normal por parte de FreeRADIUS y se procede a enviar la respuesta creada en la función. A su vez, *server_process_fragmentation_request* puede devolver el valor 2, indicando que ha finalizado el proceso de pre-autorización. En este caso, habrá que indicar a FreeRADIUS que el paquete original devuelto por esta función es válido y se procederá a dárselo a FreeRADIUS para que lo procese.

En definitiva, nuestra implementación evitará un procesamiento normal de los paquetes recibidos por parte del núcleo de FreeRADIUS si existe fragmentación. Se evitará que procese los paquetes recibidos que tengan que ver con fragmentación, hasta que este proceso haya terminado y el paquete original haya sido recompuesto, que es cuando debe procesarlo FreeRADIUS.

4.3.2.5 Gestión de datos de pre-autorización y post-autorización

Para finalizar, queda explicar cómo se generan los datos del flujo de post-autorización en el servidor y cómo se almacenan los recibidos en el flujo de pre-autorización. En concreto, vamos a implementar la posibilidad en la que el servidor haya generado durante o después del proceso de autenticación una sentencia SAML que quiera enviar al cliente. Aunque, como comentamos en la sección 4.2.3, es una solución genérica para cualquier tipo de datos que queramos enviar.

Para ello creamos un módulo de FreeRADIUS, en el que se lea una sentencia SAML de un fichero y se añada en atributos al paquete RADIUS, simulando que se ha generado por el servidor en la etapa de autenticación.

Nuestro módulo se llamará *rlm_saml*. Será necesario crear una carpeta con diferentes ficheros, por ejemplo un Makefile para compilarlo, diferentes ficheros para configurarlo y de apoyo para compilar el módulo, y también un fichero llamado de la misma forma que el módulo que contendrá el código en C. Su nombre es *rlm_saml.c* y deberá seguir una estructura para que sea compatible con FreeRADIUS.

En primer lugar crearemos la estructura de datos que utilizará el módulo. Se utilizará para almacenar en ella el nombre del fichero que contendrá el SAML a leer.

```
typedef struct rlm_saml_t {
   char* filename;
} rlm_saml_t;
```

A su vez, el módulo debe tener unos métodos obligatorios para que sea compatible con FreeRADIUS. Estos métodos son:

- *saml_instantiate*. Es necesario ya que se encarga de instanciar (inicializar) el módulo cuando FreeRADIUS es iniciado. Se carga la configuración alojada en el fichero de configuración del módulo para inicializar la estructura de datos que acabamos de comentar. Ese fichero se encuentra en la carpeta de configuración de FreeRADIUS (habitualmente en /usr/local/etc/raddb).

- *saml_detach*. Su función es liberar los recursos ocupados por el módulo al parar la ejecución del servidor.

Existe una estructura que es de vital importancia y debe aparecer en cualquier módulo para FreeRADIUS. Podemos verla a continuación:

```
module_t rlm_saml = {
   RLM_MODULE_INIT,
   "saml",
   RLM_TYPE_THREAD_SAFE,         /* type */
   saml_instantiate,             /* instantiation */
   saml_detach,                  /* detach */
   {
      NULL,                      /* authentication */
      saml_readAttribute,        /* authorization */
      NULL,                      /* preaccounting */
      NULL,                      /* accounting */
      NULL,                      /* checksimul */
      NULL,                      /* pre-proxy */
      NULL,                      /* post-proxy */
      saml_insertattribute       /* post-auth */
   }
```

En ella se configura el nombre de los métodos de *instanciación* y *detach*. También configuramos el nombre de los métodos que se ejecutarán en cada una de las etapas de FreeRADIUS. Estas etapas son, por ejemplo, autenticación, accounting, autorización, post-autorización, etc.

En nuestro caso, en esta estructura vamos a indicar el nombre de los dos métodos que creamos para tratar los SAML recibidos en los dos flujos. De esta forma, cuando se ejecute la etapa correspondiente, se llamará al método del módulo indicado en esta estructura (NULL indica que no está activo). Estas funciones recibirán en ambos métodos el paquete RADIUS original obtenido después del flujo de fragmentación.

Los nombres de los métodos son:

- *saml_readAtribute*: se ejecuta justo al acabar el flujo de pre-autorización. Se encargará de comprobar si el paquete RADIUS contiene atributos *SAML-AAA-Assertion*. En caso de existir se recuperan dichos atributos del paquete recibido del flujo de pre-autorización y los guarda en un fichero.

- *saml_insertattribute*: se ejecuta al finalizar el proceso de autenticación y es el punto de partida del flujo de post-autorización. Se encarga de añadir al paquete de respuesta el *SAMLResponse* para el cliente. Para ello, añadirá el contenido del SAML almacenado en el fichero en atributos *SAML-AAA-Assertion* al paquete.

Con esto ya quedaría finalizada la creación del módulo. Lo único que faltaría es añadir este módulo a la lista de módulos estables de FreeRADIUS, y por lo tanto, los que serán cargados al iniciar la ejecución del servidor.

4.3.3 Implementación en el cliente

En este punto vamos a explicar cómo hemos llevado a cabo la implementación tanto en nuestro cliente básico como en la implementación de ABFAB/Moonshot. En primer lugar crearemos un cliente básico para probar la implementación y el diseño del draft con *radclient* y *radeapclient,* que son clientes que se encuentran disponibles en FreeRADIUS. Una vez validada esta implementación, integramos la fragmentación en el cliente RADIUS de ABFAB/Moonshot.

4.3.3.1 Funciones y estructuras de datos para la fragmentación

En primer lugar vamos a explicar la estructura de datos que hemos decidido implementar en el cliente.

Su nombre es *struct client_frag_state_t* y tiene la misma finalidad que la estructura del servidor. A su vez, añade algunos campos que no había antes. Por ejemplo:

```
typedef struct client_frag_state_t {
  int initiator;
  int id;
  VALUE_PAIR* username;
  VALUE_PAIR*list;
  struct client_frag_state_t *next;
} client_frag_state_t;
```

Apreciamos campos similares al anterior, como *initiator* y *list*, teniendo la misma finalidad que en la estructura del servidor. El campo *id* se usará para la misma funcionalidad que el campo *state* de la estructura del servidor. Nos ayudará a vincular los paquetes recibidos con un proceso de fragmentación en curso. Por último, el atributo *username* lo utilizaremos para almacenar el nombre del usuario, ya que debe ir en todos los paquetes como vimos en la sección 4.2.

Esta estructura será la que se utilice en el cliente para almacenar los estados de fragmentación con los atributos del paquete original.

El cliente también hará uso de la función *fragmentation_required* vista en el apartado 4.3.2.2. Por otro lado, queda aún por explicar las tres funciones principales que

implementan el soporte de los dos flujos en los que puede existir fragmentación, siendo muy similares a las vistas en el servidor.

4.3.3.1.1 Pre-autorización

La primera de ellas se encarga de comprobar si un paquete que va a ser enviado por el cliente necesita ser fragmentado o no en el flujo de pre-autorización. Se llama *client_create_fragmentation_request* y la cabecera de la misma es:

```
 int client_create_fragmentation_request(RADIUS_PACKET *packet, char *secret)
{
…
}
```

Recibe como parámetro el paquete original que va a ser enviado. Se comprobará si es necesario fragmentar, y en caso de serlo, se procederá a crear el *chunk* y almacenar los atributos restantes. El parámetro *secret* se añade por la misma razón que en la función del servidor, es utilizado por la función *fragmentation_required*.

El procedimiento interno de la función es similar a su homóloga *server_create_fragmentation_request*. La única diferencia es que en esta función del cliente se utilizará el estado de fragmentación del cliente y que no será necesario extraer los atributos *State* y *Service-Type* para añadirlos al final de la lista de atributos como si hacíamos en la función del servidor, ya que no son generados por el cliente.

La próxima función es la encargada de controlar si un mensaje recibido es la respuesta a un *chunk* enviado por el cliente en el flujo de pre-autorización.

```
int client_process_fragmentation_response(RADIUS_PACKET *reply, RADIUS_PACKET
*request)
{
…
}
```

Recibe como parámetros el paquete *reply* que es el recibido por la red y *request* que es sobre el que construiremos el nuevo *chunk*. Su funcionamiento es similar a su función homóloga *server_process_fragmentation_response* del servidor. La única variación se encuentra cuando se recibe una respuesta a un *chunk* enviado anteriormente, ya que se añadirá al paquete *request* una copia del atributo *State* recibido.

4.3.3.1.2 Post-autorización

Por último, nos encontramos con la función que comprueba si el paquete recibido es un *chunk*. Se llama *client_process_fragmentation_request*, y en caso afirmativo almacenará los atributos recibidos y construirá la respuesta. Cabe destacar que tendrá utilidad en el flujo de post-autorización, ya que será ahí cuando un cliente pueda recibir *chunks*.

```
int client_process_fragmentation_request(RADIUS_PACKET *reply_packet,
RADIUS_PACKET *request_packet)
{
…
}
```

Al igual que la anterior, recibe como parámetro un *reply_packet* que contiene el que se acaba de recibir por la red, y *request_packet* que será sobre el que construiremos la respuesta.

De nuevo, esta función es similar a su homóloga en el servidor *server_process_fragmentation_request.* Sólo variará en algunos detalles. Por ejemplo, a la hora de recibir el último *chunk* no se construirá una respuesta para el otro extremo, como si se hacia en el servidor. Por otro lado, tanto a la hora de recibir el último *chunk*, uno intermedio o el primero, cambiarán los atributos que son almacenados en relación a la función del servidor. En este caso, los atributos que se han almacenado en función del *chunk* recibido está explicado en la sección 4.2.1.

4.3.3.2 Radclient y radeapclient

Para llevar a cabo la implementación de un cliente de prueba básico nos basamos en dos clientes que incorpora FreeRADIUS, *radclient* y *radeapclient*. *Radclient* será utilizado para dar soporte al flujo de pre-autorización. Por otro lado, *radeapclient* será el encargado de realizar la autenticación EAP del usuario y posteriormente dará soporte al flujo de post-autorización. Se realiza por separado para aprovechar los dos clientes que ya incorpora FreeRADIUS.

Todas las funciones descritas en la sección anterior estarán en el fichero *radius.c*, como se ha comentado anteriormente. El siguiente paso es indicar que cambios hemos llevado a cabo en los clientes *radclient* y *radeapclient* para dar soporte a los flujos descritos en la sección 4.2.2.

En primer lugar veremos *radclient*. Creamos la función *readSAMLAttr*, que es la encargada de leer de un fichero los datos SAML que vamos a incluir en nuestro paquete para el flujo de pre-autorización. En dicha función, haremos lo mismo que en el módulo *rlm_saml* explicado en el punto 4.3.2.5. Las funciones que vamos a modificar para integrar el soporte de fragmentación son:

- *radclient_init*. En esta función se inicializa el cliente y se construye el primer *Access Request*. Vamos a añadir una llamada a *readSAMLAttr* para añadir al paquete los datos de pre-autorización.

- *send_one_packet*. En esta función se envía el paquete creado en *radclient_init*. Vamos a modificarla para añadir la llamada a la función *client_create_fragmentation_request* para comprobar si el mensaje debe ser fragmentado.

- *recv_one_packet*. En ella se recibe un paquete RADIUS. Añadiremos la llamada a la función *client_process_fragmentation_request* y también a la función *client_process_fragmentation_response* para comprobar que tipo de paquete se ha recibido. En función del tipo de paquete se modificará la variable *more_chunks* para indicar si el proceso continúa o no.

- *Main*. Este método contiene un bucle que llamará a las dos funciones anteriores. Añadiremos una condición a dicho bucle para que realice una nueva iteración si la variable *more_chunks* indica que hay que continuar.

El siguiente paso es añadir el flujo de post-autorización a *radeapclient*. Hemos modificado la función *send_request_frag*, que se encarga de enviar y recibir paquetes RADIUS. Dicha función contiene un bucle, y dentro de él se envían y reciben los

paquetes. Así pues añadiremos la llamada a *client_create_fragmentation_request* antes de enviar un paquete. Por otro lado, después de recibir un paquete, llamaremos del mismo modo que en *radclient* a *client_process_fragmentation_request* y *client_process_fragmentation_response.* En función del tipo de paquete recibido se realizará, o no, otra iteración del bucle.

Por último, al finalizar el proceso de fragmentación de post-autorización incluimos un código que guarda los datos recibidos en el flujo de post-autorización en un fichero.

4.3.3.3 ABFAB/Moonshot

Para probar la implementación en un escenario más realista hemos decidido integrar la fragmentación en el cliente RADIUS de ABFAB/Moonshot. No obstante, a la hora de implementarlo, vimos que existía un pequeño problema. Para realizar la implementación en *radclient* o *radeapclient* utilizábamos algunas funciones que proporcionaba FreeRADIUS. Esas funciones se encuentran dentro de la librería *libradius*, y ABFAB/Moonshot tiene implementado RADIUS mediante una librería distinta, llamada *libradsec*. Realmente existen las mismas funciones con la misma funcionalidad, pero está estructurado de forma diferente, utilizando distintos nombres, etc. Por esa razón, tuvimos que adaptar las funciones de fragmentación de cliente para utilizar las equivalentes de *libradsec*.

Para integrar la fragmentación, primero tendremos que modificar la librería *libradsec* para que incorpore las llamadas a nuestras funciones que gestionan la fragmentación. Por otro lado, tendremos que añadir al inicio del proceso de autenticación de ABFAB/Moonshot utilizando EAP, el código que nos permita añadir los datos SAML a enviar en el flujo de pre-autorización. De esta forma, sabemos que habrá que modificar, por un lado la librería *libradsec* y por otro modificar la librería GSS-EAP.

En primer lugar, tuvimos que modificar el fichero *dictionary.txt* dentro de la librería *libradsec*. Contiene la definición de los atributos RADIUS, y en ella vamos a incluir los nuevos atributos que se definen para fragmentación. Incluiremos tanto los atributos *Frag-Status* y *Proxy-State-Len* como el nuevo valor para el atributo *Service-Type*.

También decidimos modificar el fichero *request.c* de *libradsec*. En dicho fichero incluiremos las llamadas a las funciones principales que gestionarán la fragmentación.

```
if(r==RSE_OK || fragmentation)
   //si todo es correcto o hay fragmentación en curso, comprobar si hay que
     fragmentar el paquete a enviar
   frag_status_send = moonshot_create_fragmentation_request(request>req_msg,
                                          frag_status_send,  *resp_msg);

    r = rs_packet_send (request->req_msg, NULL);
    if (r == RSE_OK) {
      r = rs_conn_receive_packet (request->conn, request->req_msg, resp_msg);
   if (r == RSE_OK) {
      //si el paquete se recibió correctamente se comprueba si es fragmentado
         fragmentation = moonshot_process_fragmentation(*resp_msg, request-
                              >req_msg, &frag_status_recv, frag_status_send);
         if(fragmentation) {
            //si el paquete recibido es fragmentado se reinician los timestamp
             y se continúa con la fragmentación
             continue;
         } else {
           // no es un paquete fragmentado, procesamiento normal
            break;
         }
      }
   }
```

En ABFAB/Moonshot será necesario añadir la estructura *client_frag_state*. Lo haremos en el fichero *request.c* y será similar a la explicada en la sección 4.3.3.1 a excepción de la eliminación de algunos campos que no eran necesarios almacenar en este cliente.

```
typedef struct client_frag_state_t {
    VALUE_PAIR* username;
    VALUE_PAIR* list;
} client_frag_state_t;
```

Por otro lado, nos vemos en la obligación de crear una función que elimine un atributo de una lista de atributos. La razón se debe porque al hacer la integración del soporte de fragmentación en *libradsec* tenemos problemas con la función que realiza esto y decidimos crearla nosotros. Se llama *delete_vp* y dada una lista de atributos y el tipo de atributo que queremos borrar, lo localiza y lo elimina (si existe).

También incluimos la función *fragmentation_required*. Por último, como funciones auxiliares, añadimos otra llamada *print_saml_received*, que se encargará de escribir en un fichero el contenido del SAML recibido en el flujo de post-autorización.

A continuación, tenemos las dos funciones claves que se encargarán de gestionar la fragmentación para los paquetes que se envían y los que se reciben. En primer lugar tenemos la función *moonshot_create_fragmentation_request*. Realizará lo mismo que su homóloga en *radclient*, haciendo uso de las funciones que ahora proporcionará *libradsec* para añadir, copiar, eliminar y etc. atributos de un paquete. También modificamos un poco la estructura de la función y los parámetros que va a recibir, ya que ahora no utilizamos la lista que almacena los estados de fragmentación.

```
client_frag_state_t* moonshot_create_fragmentation_request(struct rs_packet
*pkt, client_frag_state_t* frag_state, struct rs_packet *resp_msg)
{
…
}
```

Recibe como parámetro el paquete RADIUS a enviar (se encuentra en *rs_packet*), el paquete que contiene el último mensaje que se ha recibido (*resp_msg*) y el estado de

fragmentación (*frag_state*). Dicho estado, inicialmente tendrá valor *NULO*, pasando a tener valor si se requiere fragmentación en pre-autorización. El funcionamiento será el mismo a su homóloga *client_create_fragmentation_request*. Simplemente variará al inicio. En este caso, si existe un estado de fragmentación se añadirán al paquete a enviar los atributos restantes almacenados en el estado de fragmentación (parámetro frag_state).

La otra función que gestiona la fragmentación es la que se ejecuta cuando se recibe un paquete. Su nombre es *moonshot_process_fragmentation* y en ella comprobamos que el paquete recibido sea una respuesta a un *chunk*, un *chunk*, o un paquete normal. Esta función fusiona la funcionalidad de las dos funciones vistas en *radclient*, *client_process_fragmentation_request* y *client_process_fragmentation_response*.

```
int moonshot_process_fragmentation(struct rs_packet *pkt_received, struct
rs_packet *last_request, client_frag_state_t** fs_recv, const
client_frag_state_t* fs_send)
{
…
}
```

La función devuelve un valor entero, que será el código que indicará si hay fragmentación, si no hay, o si existe algún error. A su vez como parámetro tiene el paquete que se acaba de recibir por la red, el último paquete que enviamos por la red, y dos estados de fragmentación, el estado de fragmentación de pre-autorización y de post-autorización (debido a que tratamos esos dos casos en esta función).

Esta función es similar a *client_process_fragmentation_request* añadiendo un caso más. En ella tratábamos los posibles casos que existían al recibir un *chunk*. El caso que se añade comprueba si el paquete recibido es una respuesta a un *chunk* enviado en pre-autorización. De ser así, indicaríamos que el proceso debe continuar y se procede a enviar un nuevo *chunk* llamando a la función *moonshot_create_fragmentation_request*.

Por otro lado, existe una variación más. Ahora no se recuperan los estados de fragmentación en base al atributo *State* o al identificador del paquete. No se realiza ya que los estados de fragmentación son únicos tanto para pre-autorización como post-autorización y se pasan como parámetros en las funciones.

Con todo esto, quedaría implementada toda la funcionalidad para que ABFAB/Moonshot actúe como cliente en el proceso de fragmentación. Simplemente queda añadir a ABFAB/Moonshot el flujo de pre-autorización. Por defecto, ABFAB/Moonshot no lo incorpora, no permite enviar datos de pre-autorización al servidor antes de la autenticación EAP. No obstante, para probar los flujos definidos en la propuesta de fragmentación, lo incluímos en el código. Por lo que, antes de iniciar la autenticación EAP se puede enviar mediante RADIUS utilizando fragmentación, el contenido SAML de un fichero.

Hemos decidido incluirlo dentro del código correspondiente a GSS-EAP. Existe una clase llamada *accept_sec_context.c* que contiene la inicialización del contexto de la autenticación EAP para GSS-API. En ella, hemos incluido el código que crea el flujo de pre-autorización dentro del método que se encarga de cargar la identidad del usuario para realizar la autenticación EAP. De esta forma, ya tendríamos disponible el *User-Name* para incluirlo en los mensajes de pre-autorización.

La función en el que incluimos todo esto se llama *importInitiatorIdentity*. Justo después de tener la identidad del usuario leemos del fichero el SAML (si está vacío se omite la pre-autorización) y se añade al paquete utilizando el atributo *SAML-AAA-Assertion*. Añadimos el atributo *User-Name* y lo enviamos llamando al ya visto método *rs_request_send* que se encargará de enviar dicho paquete e iniciar el proceso de fragmentación si fuera necesario.

5. Validación de la implementación

En esta sección vamos a mostrar unas capturas que muestran el funcionamiento del escenario propuesto. Para ello hemos utilizado tres máquinas virtuales montadas sobre VirtualBox versión 4.3.10. Cada máquina virtual tiene Ubuntu 12.04 como sistema operativo. Una de estas máquinas contiene el software ABFAB/Moonshot que actuará como cliente en la fragmentación y tendrá la dirección IP *192.168.1.11*. Otra máquina actuará como servidor FreeRADIUS y tendrá la dirección IP *192.168.1.12*. Por último incluiremos un tercer equipo que actuará como *proxy legacy*, teniendo un servidor FreeRADIUS sin soporte de fragmentación. Su dirección IP será *192.168.1.13*. En la **Figura 15** podemos ver el escenario con los equipos.

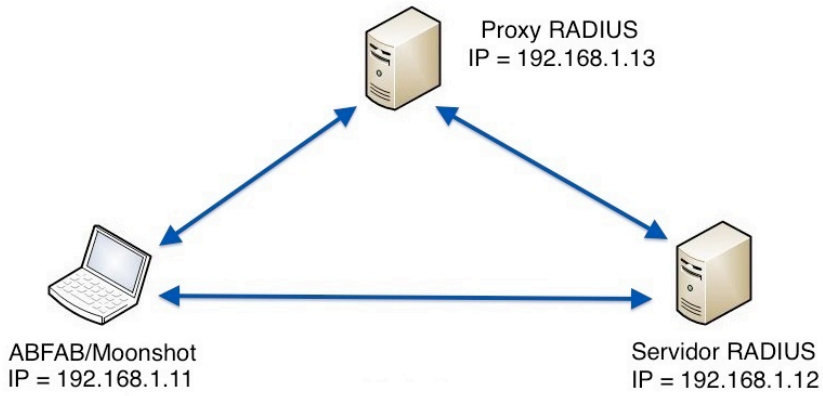

Figura 15: Escenario para la validación de la implementación

Hemos añadido un proxy para demostrar que el flujo de paquetes fragmentados es totalmente transparente a proxies, tal y como se establecía en los objetivos del draft que implementamos en este trabajo, y que por lo tanto, deben ser satisfechos.

Vamos a mostrar dos ejemplos que se basarán en la autenticación del usuario a través de ABFAB/Moonshot. Usaremos EAP-TLS como mecanismo de autenticación. Por simplicidad y para verlo más claro, un primer ejemplo tendrá simplemente el flujo de pre-autorización con el proxy actuando como elemento intermedio para demostrar que la solución es transparente a proxies. De esta forma, también podremos ver el uso del atributo *Proxy-State-Len* por parte del cliente. Seguidamente, veremos el segundo ejemplo que simplemente tendrá el flujo de post-autorización, sin el proxy como elemento intermedio.

En la instalación y configuración del escenario hay que modificar algunos ficheros de los distintos paquetes software para que funcione correctamente. En primer lugar, en ABFAB/Moonshot tendremos que añadir la dirección IP a la que enviará los paquetes RADIUS. Para ello modificamos el fichero *radsec.conf* alojado en la carpeta *etc* dentro del directorio donde está compilado ABFAB/Moonshot.

Por otro lado, en el servidor FreeRADIUS con fragmentación hay que añadir tanto el usuario y su contraseña, como el cliente RADIUS del que aceptaremos paquetes entrantes. Para ello, en la carpeta donde se encuentran los ficheros de configuración de FreeRADIUS *(/usr/local/etc/raddb/)* modificamos el fichero *users* añadiendo el usuario de prueba *moonshot@fragtest.es* con contraseña *moonshot*.

En la misma carpeta, en el fichero *clients.conf* añadimos que acepte paquetes desde la dirección IP de ABFAB/Moonshot (cuando incluyamos el proxy cambiaremos esa dirección IP por la del proxy).

Por último, en el proxy hay que modificar tanto el fichero *clients.conf* para aceptar paquetes desde ABFAB/Moonshot, como el fichero *proxy.conf*. Este fichero es necesario para que reenvíe los paquetes hacia el servidor FreeRADIUS con fragmentación.

A continuación podremos ver ambos flujos funcionando de forma correcta por separado, siendo ambos con ABFAB/Moonshot. Para ello nos hemos basado en los flujos descritos en el draft de AAA, SAML Y ABFAB/Moonshot [46], enviando un *SAMLAuthenticationRequest* en pre-autorización y un *SAMLAssertion* en el flujo de post-autorización.

5.1 Flujo de pre-autorización

14 1.839626	192.168.1.13	192.168.1.12	RADIUS	547	Access-Request(1) (id=183, l=505)
15 1.840158	192.168.1.12	192.168.1.13	RADIUS	108	Access-Accept(2) (id=183, l=66)
17 1.842049	192.168.1.13	192.168.1.12	RADIUS	478	Access-Request(1) (id=221, l=1916)
18 1.842288	192.168.1.12	192.168.1.13	RADIUS	108	Access-Accept(2) (id=221, l=66)
19 1.852360	192.168.1.13	192.168.1.12	RADIUS	374	Access-Request(1) (id=182, l=332)
20 1.854010	192.168.1.12	192.168.1.13	RADIUS	90	Access-Accept(2) (id=182, l=48)
21 1.855358	192.168.1.13	192.168.1.12	RADIUS	139	Access-Request(1) (id=74, l=97)
22 1.855838	192.168.1.12	192.168.1.13	RADIUS	109	Access-Challenge(11) (id=74, l=67)
23 1.936839	192.168.1.13	192.168.1.12	RADIUS	371	Access-Request(1) (id=191, l=329)
24 1.939630	192.168.1.12	192.168.1.13	RADIUS	1135	Access-Challenge(11) (id=191, l=1093)
28 2.022087	192.168.1.13	192.168.1.12	RADIUS	146	Access-Request(1) (id=38, l=104)
29 2.022647	192.168.1.12	192.168.1.13	RADIUS	1135	Access-Challenge(11) (id=38, l=1093)
31 2.102607	192.168.1.13	192.168.1.12	RADIUS	146	Access-Request(1) (id=141, l=104)
32 2.102943	192.168.1.12	192.168.1.13	RADIUS	648	Access-Challenge(11) (id=141, l=606)
33 2.186115	192.168.1.13	192.168.1.12	RADIUS	280	Access-Request(1) (id=15, l=238)
34 2.190941	192.168.1.12	192.168.1.13	RADIUS	172	Access-Challenge(11) (id=15, l=130)
38 2.268850	192.168.1.13	192.168.1.12	RADIUS	252	Access-Request(1) (id=174, l=210)
39 2.271002	192.168.1.12	192.168.1.13	RADIUS	182	Access-Challenge(11) (id=174, l=140)
40 2.348890	192.168.1.13	192.168.1.12	RADIUS	284	Access-Request(1) (id=38, l=242),
41 2.349514	192.168.1.12	192.168.1.13	RADIUS	226	Access-Accept(2) (id=38, l=184)

Figura 16: Mensajes RADIUS en el flujo de pre-autorización con Moonshot

En la **Figura 16** observamos todo el flujo de mensajes que tiene lugar en una autenticación mediante ABFAB/Moonshot. En ella, los seis primeros mensajes, que se encuentran rodeados en el recuadro rojo, se corresponden con el intercambio del flujo de pre-autorización. Dentro de los *Access Request* tendremos los datos SAML que quiere enviar ABFAB/Moonshot hacia el servidor FreeRADIUS. Seguidamente, vemos el resto de mensajes, en los que se produce todo el intercambio de mensajes para la autenticación EAP. El intercambio ha sido capturado en el servidor FreeRADIUS, por ello la dirección IP origen del primer paquete se corresponde con la del proxy, ya que en este escenario es utilizado.

En la **Figura 17,** observamos el contenido del primer chunk que envía ABFAB/Moonshot hacia el servidor. En él podemos apreciar, rodeado en rojo, como se envían los atributos *Frag-Status* (atributo 224) con *valor 2* y *Service-Type* con *valor 200* tal y como lo explicamos en la sección 4.2.2. También apreciamos los atributos que se han añadido al paquete y que contienen el SAML, marcados con el número de atributo 26, ya que es un atributo perteneciente a un diccionario externo (diccionario *Ukerna*). Por último observamos como existe un proxy en el camino que ha seguido el paquete, ya que tenemos el atributo *Proxy-State*.

```
Attribute Value Pairs
▷ AVP: l=14  t=User-Name(1): @fragtest.es
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=6  t=Unknown-Attribute(224): 00000002
▷ AVP: l=6  t=Service-Type(6): Unknown(200)
▷ AVP: l=18  t=Message-Authenticator(80): 41592faab29c9c948646833c741cf449
▷ AVP: l=6  t=NAS-IP-Address(4): 192.168.1.11
▷ AVP: l=3  t=Proxy-State(33): 30
```

Figura 17: Contenido del chunk de pre-autorización

En la respuesta que realiza el servidor, observamos como responde pidiendo el resto de datos (*Frag-Status* con *valor 3*), añadiendo el atributo *State* y lo que es más importante, marcando en el atributo *Proxy-State-Len* (atributo 220) el número de bytes que ocupan los atributos *Proxy-State* que se incluyen por el camino debido a los proxies. A su vez, observamos el atributo *Service-Type* con valor 200 (Additional-Authorization). Todo esto lo podemos comprobar en la **Figura 18**.

```
Attribute Value Pairs
▷ AVP: l=6  t=Unknown-Attribute(224): 00000003
▷ AVP: l=6  t=Unknown-Attribute(220): 00000003
▷ AVP: l=18  t=Message-Authenticator(80): a720396b8f6f59d61bbe60e3a05d6771
▷ AVP: l=7  t=State(24): 4672616730
▷ AVP: l=6  t=Service-Type(6): Unknown(200)
▷ AVP: l=3  t=Proxy-State(33): 30
```

Figura 18: Contenido de la respuesta al chunk de pre-autorización

Por último, mostramos un ejemplo del último chunk enviado por el cliente en este flujo. En él podemos comprobar como no se incluye ningún atributo relacionado con fragmentación, y simplemente irán los datos que quedan por transportar y el atributo *State*. Lo podemos ver en la **Figura 19**.

```
Attribute Value Pairs
▷ AVP: l=14  t=User-Name(1): @fragtest.es
▷ AVP: l=7  t=State(24): 4672616731
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom
▷ AVP: l=48  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom E
▷ AVP: l=18  t=Message-Authenticator(80): 904023efa03edab9112af4ab018f2704
▷ AVP: l=6  t=NAS-IP-Address(4): 192.168.1.11
▷ AVP: l=3  t=Proxy-State(33): 30
```

Figura 19: Contenido del último chunk de pre-autorización

Por otro lado, podemos observar como se ha empleado un tamaño de chunk conservador en el primer paquete respecto de los paquetes posteriores. La razón se debe a que no sabemos el número de proxies en el camino y por lo tanto el paquete puede ser descartado, especialmente debido al numero de atributos *Proxy-State* que sean añadidos. Es por ello, que el segundo chunk tiene un tamaño mayor, gracias a que el cliente ya conoce el espacio que hay que dejar reservado para estos atributos. También resaltar que ha quedado comprobado que la solución es totalmente transparente a proxies. Y a su vez, que la autenticación ha sido exitosa, ya que el último mensaje recibido es un *Access Accept* que garantiza el acceso.

5.2 Flujo de post-autorización

En el flujo de post-autorización, en la **Figura 20**, observamos que no hay ningún mensaje de pre-autorización, se ejecuta directamente la autenticación EAP. A su vez, rodeado en rojo, vemos el primer *Access Accept* correspondiente al flujo de post-autorización que contiene los primeros datos fragmentados.

```
27 3.672512   192.168.1.11   192.168.1.12   RADIUS   130 Access-Request(1) (id=0, l=88)
28 3.673042   192.168.1.12   192.168.1.11   RADIUS   106 Access-Challenge(11) (id=0, l=64)
29 3.752644   192.168.1.11   192.168.1.12   RADIUS   362 Access-Request(1) (id=0, l=320)
30 3.755898   192.168.1.12   192.168.1.11   RADIUS  1132 Access-Challenge(11) (id=0, l=1090)
31 3.836904   192.168.1.11   192.168.1.12   RADIUS   137 Access-Request(1) (id=0, l=95)
32 3.839050   192.168.1.12   192.168.1.11   RADIUS  1132 Access-Challenge(11) (id=0, l=1090)
33 3.916863   192.168.1.11   192.168.1.12   RADIUS   137 Access-Request(1) (id=0, l=95)
34 3.917296   192.168.1.12   192.168.1.11   RADIUS   645 Access-Challenge(11) (id=0, l=603)
36 3.996443   192.168.1.11   192.168.1.12   RADIUS   271 Access-Request(1) (id=0, l=229)
37 3.997938   192.168.1.12   192.168.1.11   RADIUS   169 Access-Challenge(11) (id=0, l=127)
38 4.076809   192.168.1.11   192.168.1.12   RADIUS   243 Access-Request(1) (id=0, l=201)
39 4.077337   192.168.1.12   192.168.1.11   RADIUS   179 Access-Challenge(11) (id=0, l=137)
42 4.156376   192.168.1.11   192.168.1.12   RADIUS   275 Access-Request(1) (id=0, l=233)
44 4.157097   192.168.1.12   192.168.1.11   RADIUS   375 Access-Accept(2) (id=0, l=1813)
45 4.158294   192.168.1.11   192.168.1.12   RADIUS   113 Access-Request(1) (id=0, l=71)
47 4.158909   192.168.1.12   192.168.1.11   RADIUS   455 Access-Accept(2) (id=0, l=1893)
48 4.159526   192.168.1.11   192.168.1.12   RADIUS   113 Access-Request(1) (id=0, l=71)
50 4.159802   192.168.1.12   192.168.1.11   RADIUS    62 Access-Accept(2) (id=0, l=1500)
```

Figura 20: Mensajes RADIUS en el flujo de post-autorización con Moonshot

En la **Figura 21** observamos el contenido del primer *Access Accept* enviado por el servidor. Ahora el iniciador es el servidor FreeRADIUS y vemos como añade al paquete los atributos de fragmentación al igual que lo hacía anteriormente el cliente. También observamos que aparece un atributo EAP que contiene el *EAP-Success* y hay otros dos atributos específicos del diccionario Microsoft que los utiliza ABFAB/Moonshot para enviar las claves de envío y recepción de mensajes a través de este cliente.

```
Attribute Value Pairs
▷ AVP: l=58  t=Vendor-Specific(26) v=Microsoft(311)
▷ AVP: l=58  t=Vendor-Specific(26) v=Microsoft(311)
▷ AVP: l=6  t=EAP-Message(79) Last Segment[1]
▷ AVP: l=18  t=Message-Authenticator(80): dfa71382299be930b6c8b11039a1a651
▷ AVP: l=14  t=User-Name(1): @fragtest.es
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=108  t=Vendor-Specific(26) v=JANET(UK) (previous was 'UKERNA (United Kingdom Education
▷ AVP: l=7  t=State(24): 4672616730
▷ AVP: l=6  t=Unknown-Attribute(224): 00000002
▷ AVP: l=6  t=Service-Type(6): Unknown(200)
```

Figura 21: Contenido del primer chunk de post-autorización

Por último, vemos la respuesta que realiza ABFAB/Moonshot al paquete enviado por el servidor. Responde tal y como hemos explicado anteriormente, enviando una copia del atributo *State*, el atributo *Frag-Status* con *valor 3*, etc. En la **Figura 22** lo podemos observar.

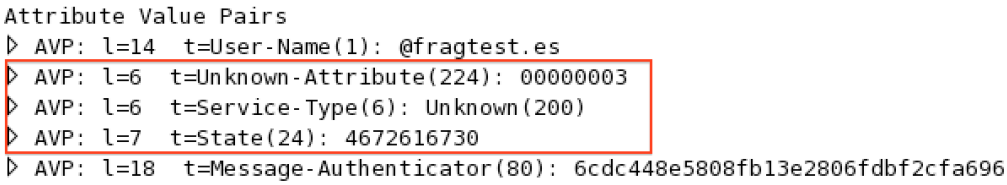

Figura 22: Contenido de la respuesta al chunk de post-autorización

Finalmente, resaltar que el servidor envía todos los paquetes fragmentados como *Access Accept* y que después de enviar el último *chunk* no se recibe respuesta por parte del cliente, ya que ABFAB/Moonshot recompone el mensaje original y lo procesa mostrando un mensaje que indica que ha sido validada la autenticación, tal y como podíamos ver en el atributo *EAP-Success* del primer *chunk*.

6. Conclusión y vías futuras

A lo largo de este trabajo se ha implementado por completo el diseño especificado en el draft para el intercambio de datos de autorización de gran tamaño sobre RADIUS. En concreto, hemos implementado el flujo especificado en la propuesta que permite la fragmentación de paquetes RADIUS que contienen este tipo de datos. Esta propuesta está en los últimos pasos para ser estándar internacional, y esta implementación será utilizada para demostrar el correcto diseño y funcionamiento de la misma.

La creación de este draft ha sido posible gracias a la colaboración de la Universidad de Murcia, en especial de Alejandro Pérez Méndez, Rafael Marín López, Gabriel López Millán, con Alan Dekok, (creador de FreeRADIUS), Janet y Telefónica I+D.

Hemos conseguido implementar y probar la funcionalidad diseñada en el draft, así como realizar una prueba enviando datos SAML. A su vez, hemos integrado dicha funcionalidad de fragmentación en una arquitectura utilizada mundialmente como es ABFAB/Moonshot y que va a permitir crear un escenario de prueba basado en casos de uso reales. También hemos probado la implementación en un escenario que contiene proxies legacy, demostrando que se cumple correctamente uno de los objetivos diseñados en el draft.

Por otro lado, durante la implementación del diseño encontramos un fallo en la especificación de la propuesta. Dicho fallo tenía lugar durante el flujo de pre-autorización y provocaba el rechazo de los paquetes recibidos por el servidor. Pudimos averiguar que la causa del mismo era provocada porque los paquetes enviados por el cliente RADIUS no contenían ni el atributo *User-Name* ni ninguno relacionado con un proceso de autenticación. Esto provocaba un error en el servidor y el rechazo del paquete. Finalmente, se optó por incluir en el draft y en la implementación, el envío del atributo *User-Name* en todos los paquetes enviados por el cliente en el flujo de pre-autenticación. A su vez, se implementó en el servidor como añadido al soporte de fragmentación, la validación de todos los paquetes recibidos pertenecientes al flujo de pre-autorización. Esta solución, muy discutida dentro del grupo de trabajo RADEXT del IETF, viola el RFC 2865, pero fue consensuada y aprobada, quedando pendiente de una revisión posterior.

A su vez, este trabajo está incluido en un proyecto a nivel europeo llamado CLASSe mediante el cual se quiere integrar un servicio de cloud a un escenario con acceso federado y Single Sign-On con tecnología ABFAB/Moonshot. En dicho escenario existe la necesidad de enviar datos de autorización en SAML desde el servidor de autenticación hasta el servicio de cloud. Este escenario es similar al que hemos implementado en este trabajo, siendo ABFAB/Moonshot en ambos casos el punto de conexión entre el servidor de autenticación y la aplicación.

Por último, esta implementación estará disponible en la web del proyecto, tanto el código fuente como una imagen ISO que contiene todo el escenario desplegado y en correcto funcionamiento. A su vez, se ha creado un *branch* en el repositorio de ABFAB/Moonshot para que cualquier persona pueda añadir esta funcionalidad a su versión de ABFAB/Moonshot y probar la fragmentación de paquetes RADIUS.

En relación a vías futuras, este trabajo queda abierto a posibles mejoras en el diseño del draft, así como la implementación de *proxies updated*, la integración del diseño de fragmentación en clientes diferentes al utilizado en este trabajo y al mantenimiento de la implementación. Por otro lado, actualmente se está trabajando en la preparación de una guía de instalación que permita dejar todo lo desarrollado en este trabajo de forma pública a través del proyecto CLASSe. Así, cualquier persona que lo estime oportuno podrá descargar esta implementación e integrar el soporte de fragmentación en su propio servidor RADIUS o ABFAB/Moonshot para probarlo y utilizarlo.

7. Referencias

[1] Entrada Wikipedia Single Sign-On (SSO) → http://es.wikipedia.org/wiki/Single_Sign-On

[2] C. Rigney, S. Willens, A. Rubens and W. Simpson. Remote Authentication Dial In User Service (RADIUS). IETF RFC 2865 → http://tools.ietf.org/html/rfc2865

[3] ABFAB WorkGroup of IETF → http://tools.ietf.org/wg/abfab/

[4] Extensible Authentication Protocol Tunneled Transport Layer Security (EAP-TTLS), IETF RFC 5281 → https://tools.ietf.org/html/rfc5281

[5] Password Authentication Protocol (PAP), IETF RFC 1334, Punto 2 → https://tools.ietf.org/html/rfc1334

[6] Challenge Handshake Authentication Protocol (CHAP), IETF RFC 1994 → http://tools.ietf.org/html/rfc1994

[7] Extensible Authentication Protocol (EAP), IETF RFC 3748 → http://tools.ietf.org/html/rfc3748

[8] C. De Laat, G. Gross, L. Gommans, J. Vollbrecht y D. Spence. Generic AAA Architecture, IETF RFC 2903 → http://tools.ietf.org/html/rfc2903

[9] OASIS. Security Assertion Markup Language(SAML) V2.0 Technical Overview → https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf

[10] Janet. Project Moonshot → https://community.ja.net/groups/moonshot

[11] Alan DeKok. The FreeRADIUS Project → http://FreeRADIUS.org

[12] Josh Howlett, Sam Hartman, Hannes Tschofenig, Eliot Lear. Application Bridging for Federated Access Beyond Web (ABFAB), IETF draft → https://tools.ietf.org/html/draft-ietf-abfab-arch-12

[13] C. Rigney. RADIUS Accounting. IETF RFC 2865→ https://tools.ietf.org/html/rfc2866

[14] J. Postel. User Datagram Protocol (UDP), IETF RFC 768 → http://www.ietf.org/rfc/rfc0768.txt

[15] M. Wahl, T. Howes y S. Kille. Lightweight Directory Access Protocol (LDAP), IETF RFC 2251 → http://www.ietf.org/rfc/rfc2251.txt

[16] MySQL 5.7 → http://dev.mysql.com/doc/refman/5.7/en/mysql-nutshell.html

[17] VLAN, artículo Wikipedia → http://es.wikipedia.org/wiki/VLAN

[18] R. Rivest. The MD5 Message-Digest Algorithm (MD5), IETF RFC 1321 → http://www.ietf.org/rfc/rfc1321.txt

[19] S. Frankel, S. Krishnan. IP Security (IPsec), IETF RFC 6071 → http://tools.ietf.org/html/rfc6071

[20] T. Dierks, E. Rescorla. The Transport Layer Security (TLS) Protocol, IETF RFC 5246 → http://tools.ietf.org/html/rfc5246

[21] Vocal Technologies. EAPoL – Extensible Authentication Protocol over LAN Software → http://www.vocal.com/secure-communication/eapol-extensible-authentication-protocol-over-lan/

[22] L. Blunk, J. Vollbrecht. PPP Extensible Authentication Protocol, EAP-MD5. IETF RFC 2284 → http://tools.ietf.org/html/rfc2284#section-3.4

[23] H. Haverinen, J. Salowey. Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM), IETF RFC 4186 → http://tools.ietf.org/html/rfc4186

[24] 802.1X, artículo Wikipedia → http://es.wikipedia.org/wiki/IEEE_802.1X

[25] W3C. Extensible Markup Language (XML) → http://www.w3.org/XML/

[26] D. Eastlake, J. Reagle, D. Solo. W3C Group. XML-Signature, IETF RFC 3275 → http://www.ietf.org/rfc/rfc3275.txt

[27] W3C Group. XML Encryption → http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/

[28] A. Dekok, G. Weber. RADIUS Design Guidelines, IETF RFC 6158 →http://tools.ietf.org/html/rfc6158

[29] J. Linn. Generic Security Service Application Program Interface (GSS-API), IETF RFC 2743 → http://tools.ietf.org/html/rfc2743

[30] Computación en la nube (Cloud) → http://es.wikipedia.org/wiki/Computación_en_la_nube

[31] A. Perez-Mendez, R. Marin-Lopez, F. Pereniguez-Garcia, G. Lopez-Millan, D. Lopez y A. DeKok. Support of fragmentation of RADIUS packets → http://tools.ietf.org/html/draft-ietf-radext-radius-fragmentation-06

[32] S.Winter, M.Mcauley, S. Venaas y K. Wierenga. Transport Layer Security (TLS) Encryption for RADIUS, IETF RFC 6614 → http://tools.ietf.org/html/rfc6614

[33] D. Simon, B. Aboba y R. Hurst. The EAP-TLS Authentication Protocol, IETF RFC 5216 → http://tools.ietf.org/html/rfc5216

[34] D. Stanley, J. Walkery B. Aboba. Extensible Authentication Protocol (EAP) Method Requirements for Wireless LANs, IETF RFC 4017 → http://tools.ietf.org/html/rfc4017

[35] OASIS, open starndars for the information society → https://www.oasis-open.org

[36] Security Assertion Markup Language (SAML) V2.0, SAML Core → http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[37] Security Assertion Markup Language (SAML) V2.0, SAML Bindings → http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf

[38] Security Assertion Markup Language (SAML) V2.0, SAML Profiles → http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf

[39] Security Assertion Markup Language (SAML) V2.0, SAML Conformance → http://docs.oasis-open.org/security/saml/v2.0/saml-conformance-2.0-os.pdf

[40] Cookies of internet, artículo de Wikipedia → http://es.wikipedia.org/wiki/Cookie_(informática)

[41] The Internet Engineering Task Force (IETF)→ http://www.ietf.org

[42] JavaScript Object Notation (JSON)→ http://www.json.org

[43] The United Kingdom Education and Research Networking Association (UKERNA), ahora JANET→ http://www.ja.net

[44] Example of SAML Reques and SAML Response with signature → https://rnd.feide.no/2007/12/10/example_saml_2_0_request_and_response/

[45] Lenguaje C, artículo de Wikipedia → http://es.wikipedia.org/wiki/C_(lenguaje_de_programación)

[46] J. Howlett, S. Hartman. AAA, ABFAB y SAML → http://tools.ietf.org/html/draft-ietf-abfab-aaa-saml-09

[47] Class-Based Policy Language (CPL) → http://www.cisco.com/c/dam/en/us/products/collateral/ios-nx-os-software/provisioning-monitoring-management/cbpp_cpl_qos.pdf

[48] RADEXT WorkGroup of IETF → http://tools.ietf.org/wg/radext/

[49] V. Fajardo, J.Arkko, J. Loughney y G. Zorn. DIAMETER Base Protocol, IETF RFC 6733 → http://tools.ietf.org/html/rfc6733

[50] Education roaming, Eduroam → http://www.eduroam.es

[51] S. Hartman y J. Howlett. A GSS-API Mechanism for the Extensible Authentication Protocol (GSS-EAP). IETF RFC 7055 → http://tools.ietf.org/html/rfc7055

[52] Protocolo AAA (Authentication, Ahutorization y Accounting) → http://es.wikipedia.org/wiki/Protocolo_AAA

[53] J. Postel y J. Reynolds. Telnet Protocol Specification, IETF RFC 854 → http://es.wikipedia.org/wiki/Telnet

[54] J. Postel y J. Reynolds. File Transfer Protocol (FTP), IETF RFC 959 → http://tools.ietf.org/html/rfc959