Hogeschool van Amsterdam
Amsterdam University of Applied Sciences

Project From Moonshot to Mars

Research Report

Remy Bien, Ruben Bras, Marvin Hiemstra
Sebastiaan Groot, Wouter Miltenburg, Koen Veelenturf

24 June 2013
Version 1.1

NIKHEF

Management summary

The goal of the project was to secure the existing federated access to SSH[1] using the user's existing credentials. The term "federated access" means that different institutions or companies can use each other services. The RADIUS[2] protocol facilitates federated access by connecting the authentication systems of multiple institutions and allowing users to securely authenticate elsewhere using their own credentials.

In practice, much information about user permissions for different services is stored on other systems than the users home institutions authentication servers. Services like VOMS[3] and SurfConext[4] provide authorization information for specific services. In order to allow RADIUS traffic to carry these additional attributes from third-party services, we've developed the basis for a protocol specifying a set of messages that can be send as part of traditional RADIUS traffic.

In this specification, the client initiating the RADIUS authentication can add a special attribute-request message to his authentication request. This message specifies the attribute provider he wishes to use, as well as the attribute he wishes to obtain and certain attributes that might be required to use this service (such as a username and password). When the authentication request passes through a proxy RADIUS server that can provide third-party services, it injects its certificate into the request. The users home institution RADIUS server then, after authenticating the user, fills in any attributes required to use the requested attribute provider and encrypts the message using the attribute providers certificate. On its way back, the proxy RADIUS server providing the third-party service can decrypt this message using its private key, and attempt to provide the requested, additional attributes for the client.

Apart from defining the protocol that allows this communication, a prototype FreeRADIUS[5] module was developed to demonstrate the mechanism. In its current phase of development, the module can successfully inject certificates from proxy RADIUS servers, receive the attribute requests as home institution RADIUS server and send it back, with required attributes filled in, to the proxy RADIUS server. Back at the proxy RADIUS server, this message is received, the requested attributes filled in and passed back to the client.

[1] The Secure Shell (SSH) Transport Layer Protocol - http://tools.ietf.org/html/rfc4253
[2] Remote Authentication Dial In User Service (RADIUS) - http://tools.ietf.org/html/rfc2865
[3] GFD.182.3 from the OGF document series
[4] http://www.openconext.org/
[5] http://freeradius.org/

Table of contents

Introduction

This Research document was created in order to give an insight into the research done by ITopia. In the first chapter the project is described and explained. Knowledge of a number of protocols was necessary. Namely SSH, GSS-API, EAP-TTLS etc. these protocols are described in the chapter ‘Research’. In chapter ‘FreeRADIUS Module’ our proposed solution to make NIKHEF’s moonshot project more secure is described. This document is part of a set of documents, the other document describes JaNET’s Moonshot project and our experience with their solution.

Project Definition

Goal

The goal of the project was to secure the existing federated access to SSH using the user's existing credentials. The term "federated access" means that different institutions or companies can use each other services. For example, when a person is at company B and wants to login at company's B server through SSH, while you are actually working for company A, access should be granted based on access rights that are defined in a VOMS-like service. This could take some serious security related threats with it, but when you can use extra attributes to allow a user some specific right for a specific service, it is possible to defeat those threats. This should be done without making the password or password hash known anywhere but the user's instances RADIUS server, so that when someone is sniffing, the sniffer cannot know the password or derive it from a hash. During the development, the focus was to follow the security principles and the creation of a standardized solution.

The research team found out that JaNET was a possible solution but did not support the extra attributes that research team was looking for. For more information, please read the document 'Research Report JaNET'. From that moment, the research team has defined a new goal and developed a FreeRADIUS module to send extra attributes in the authorization stage of RADIUS. This module should follow security principles and should ensure that all the attributes are transmitted safely.

Project conditions and limitations

The scope of the project was to study the existing literature and presenting the findings and the development of possible solutions that will enable the user to securely connect to an OpenSSH[6] server using RADIUS and send extra attributes with it. The priority of the project, considering the time restrictions, is the delivery of solutions that can be presented and used on an international level.

[6] http://www.openssh.org/

Project history

Last year another team worked on the same project, and even though they came a long way, they did not finish the entire project. Their implementation worked up to a certain point but did not guarantee the privacy and integrity of the users' credentials.

The main goal of the last project team was to implement new features without keeping secure authentication[7] in mind; this would be done in a later stage or by another project team. They implemented federated access for OpenSSH and WebDAV[8] using one pool account where all users where mapped to when they are logged in. For example, user A that is working for company B wants to log-in an SSH-server of company C. When user A is successfully authenticated it will be mapped to the pool-account and group moonshot. There are some security considerations regarding this way of mapping users. The users are not separated of each other and anyone that is logged into the system will see all files created by the pool-account moonshot.

This year we would continue their work and either improve upon, rethink or rebuild their implementation to make sure user credentials are stored and handled securely.

During the project, we discovered that the former project team's work would not provide the solution that we were aiming for. Therefore, we began searching for other possible solutions. After several solutions and discussions with different employees from Nikhef, we agreed upon a solution to work on.

Some of the possible solutions relied on Kerberos, GSS-API, EAP-TTLS and RADIUS described in the next chapters. The current research team chose RADIUS as the possible solution. In addition to the description and design of the FreeRADIUS module, the research team will deliver a prototype of their working concept. Due this development, the project's main goal was slightly changed.

[7] In this context secure authentication is that, whenever a user wants to authenticate, all credentials will be transmitted and that a man-in-the-middle-attack is pointless. The credentials that are actually sent are always encrypted in such a manner that a man-in-the-middle attack cannot derive the original password from the received bytes.
[8] HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV) - http://tools.ietf.org/html/rfc4918

Research

This chapter provides an overview of the researched protocols and software that was considered interesting for our goal in the first stage of our research. Some of these protocols and software are actually used or not used in our module. The reason for research and why we have integrated it in our solution will be answered in this chapter.

RADIUS

What is RADIUS?

RADIUS is an acronym for "Remote Authentication Dial In User Service". It is a networking protocol providing centralized Authentication, Authorization and Accounting (AAA) for computers using a network service. This is a perfect solution for Internet Service Providers (ISPs) and independent, but collaborating, institutions. Such an example would be the academically used "eduroam"[9].
RADIUS is commonly used to allow authentication on a network, without storing the user's credentials at every location in this network. Meaning you can access, per example, RADIUS_Network1 from both LocationX and LocationY. But only Location X has access to the credentials used for the authentication. LocationX would in this case be the endpoint, or Identity Provider (IdP), and LocationY would function as a proxy.
To realize this, the RADIUS protocol uses "realms". A realm is used to find the user's home institution. Not using a realm is possible, though it is recommended in the case you are accessing the RADIUS network from anywhere other than your home institution.
A realm is a piece of text appended to the username (either in postfix, prefix or both). It does not have to be a valid domain name, though it is often used as such. Any character can be used to separate the username from the realm. In practice, prefix uses "/", and postfix uses "@". A login of a user could then look like the following: "somedomain/username@otherdomain".

When attempting to connect to a RADIUS network, three replies can be sent by the server. These replies are Access-Accept, Access-Challenge and Access-Reject.

- Access-Accept means authentication succeeded.
- Access-Reject means authentication failed.
- Access-Challenge means the server needs more information before it can resolve to one of the above two replies.

To carry data between the client and different RADIUS servers, an Attribute/Value Pair structure is used. Meaning a request or response will contain several datatypes with a set attribute name and a value for this attribute.
FreeRADIUS is a solution incorporating a RADIUS server, in addition to a BSD licensed client library, PAM library, and Apache module. It is the solution used most often when an institution wants to use a RADIUS server, and for this reason we focused our research on FreeRADIUS.

[9] https://www.eduroam.org

How is RADIUS used?

RADIUS is the carrier protocol that our module uses to send its messages through. As it plays such an important role in our module, knowledge on the possibilities and limitations of RADIUS was important.

EAP-TTLS

What is EAP-TTLS

EAP-TTLS[10] (Tunneled Transport Layer Security) is designed to provide authentication that is as strong as EAP-TLS, but it does not require that each user be issued a certificate. Instead, only the authentication servers are issued certificates. User authentication is performed by password, but the password credentials are transported in a securely encrypted tunnel established based upon the server certificates.

How is EAP-TTLS used?

Seeing that EAP-TTLS is often used in federated RADIUS networks such as eduroam, it will often be active alongside our module. This could, in some situations, provide an interesting added layer of security we could leverage to send encrypted messages with forward-security from the client to the client's home institutions RADIUS server.

GSS(API)

What is GSS-API?

GSS-API[11] (Generic Security Service API) is used designed for programs to have easy access to different security services. Rather than providing security in itself, it provides implementations of security libraries.
It is an IETF[12] standard and used to combat the problem of the many incompatible security systems that are in use.

How is GSS-API used?

GSS-API is in itself not a solution, but a framework. Because GSS-API did not directly solve any of our problems, it was not used. Instead, we did more research into one of the dominant mechanisms implementing GSS-API, Kerberos.

[10] Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0) - http://tools.ietf.org/html/rfc5281
[11] Generic Security Service Application Program Interface Version 2, Update 1 - http://tools.ietf.org/html/rfc2743
[12] The Internet Engineering Task Force - http://www.ietf.org/

Kerberos

What is Kerberos?

Kerberos[13] is a network authorization protocol working on the basis of tickets. Tickets allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. It is primarily aimed at a client-server model and provides mutual authentication.
Kerberos messages are protected against both eavesdropping and replay attacks. It utilizes both symmetric key cryptography (in which case a trusted third party is required) and optionally public-key cryptography during certain phases of the authentication.
Kerberos' disadvantages however, make it unusable for our project. A first point is a lack of scalability because every realm must trust every other realm in the network. In addition, any Kerberos-proxy standing between the client and the desired service is able to learn the session key, which could lead to spoofing and eavesdropping cases, by an intermediary realm pretending to be either the client or service.

How is Kerberos used?

Kerberos was deemed unusable for our goals during the preliminary research. The scalability made it incompatible with the federated environment of networks like eduroam. We instead opted for an implementation of JaNET's Moonshot and our own custom FreeRADIUS module.

VOMS

What is VOMS?

VOMS[14], or Virtual Organization Membership Service, is a type of account database used keep track of attributes used for authorization in a grid computing environment. VOMS was developed by the *European DataGrid* and *Enabling Grids for E-sciencE* projects.

How is VOMS used?

VOMS itself is not part of the projects scope, but its usage as attribute provider for authorization purposes would make it a valuable addition to existing RADIUS authentication. VOMS was one of the services that could be supported by the FreeRADIUS module discussed in this document.

[13] The Kerberos Network Authentication Service (V5) - http://www.ietf.org/rfc/rfc4120.txt
[14] GFD.182.3 from the OGF document series - http://www.ogf.org/documents/GFD.182.pdf

FreeRADIUS Module

This chapter describes the FreeRADIUS module that was built during this project. The single sign on solution that JaNET provides does not verify any certificates that are sent by the RADIUS server. To make it worse, the whole FreeRADIUS chain is not being verified. This can result in security weaknesses with devastating results. This FreeRADIUS module is addressing the possible security weaknesses we experienced during our investigation into JaNET's moonshot project. FreeRADIUS will still work in a RADIUS-chain, but when information is sent across the network it will be encrypted. Certificates will also be sent along with this information. When extra attributes are needed, in the event a VOMS-service will be used per example, the client can request additional attributes. By sending a certificate it can be made sure that only one server in the chain can read the information that is sent. Depending on the implementation and the type of certificate that is being used in the whole chain. Implementing this module (in a JaNET moonshot-like solution) can address the benefits of securely transmitting user credentials, requesting extra attributes for a VOMS-like service and verifying the chain of RADIUS servers.

Protocol description

In this section we will describe the workings of our module's protocol. We will start by explaini ng the message format our protocol uses and then we will explain the supported algorithms. Usi ng those pieces of information as context, we shall then describe how our module exchanges da ta between different Radius servers and the client.

Message formats

The MIME[15] entities will be wrapped in a vendor-specific AVP in the RADIUS Access-Request or Access-Accept packets. The vendor-specific AVPs are added according to RFC2865.

[15] Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies - http://www.ietf.org/rfc/rfc2045.txt

Data exchange

The exchange of data between our FreeRADIUS modules as deployed on different servers, can be divided in four different messages:

- Moonshot Request, from Client to Identity Provider (IdP)
- Moonshot Certificate, from Proxy Server to IdP
- Moonshot IdP Reply, from IdP to Proxy Server
- Moonshot Proxy Reply, from Proxy Server to Client

Most of these messages are sent in the form of Uniform resource names (URNs), all of which are quite alike, but tailored uniquely to the type of request or response they are used for.

- Client request
  - Timestamp:Service_DN:Proxy_DN:Required_Attibutes:Requested_Attributes
- IdP Response
  - Timestamp:Service_DN:IdP_DN:Provided_AVPs:Requested_Attributes
- Proxy response
  - Timestamp:Proxy_DN:Requested_AVPs

Moonshot Request is a Client request, as seen in step 1 of the Topology Diagram in the appendix. It means that the Client will send a request for additional attributes, per example a username. These would be necessary to access a service, such as an SSH server.
These attributes are requested via the earlier mentioned URN. This URN is sent as a message in the module. Messages in the module are sent as Attribute-Value Pairs as part of RADIUS packets. All messages have MIME-headers indicating their contents. The Transfer-Encoding of all messages is Base64, where the headers itself are in US ASCII.

When the Proxy Server receives this request in step 2, it will pack its Certificate as part of the RADIUS packets before sending them through. Like the Client request, this is done in an Attribute-Value Pair, and once again with a MIME-header. The header would in this case indicate we are dealing with a certificate. The certificates are sent as plaintext MIME messages in PEM format This message, which we call Moonshot Certificate, is then sent on to the IdP.

The IdP, in step 3, will read the URN to determine what Attribute-Value Pairs the Client requests, and obtains these from a VOMS-like service where they can be found. These attributes are placed in a URN. This message is signed using the Proxy's Certificate, and encrypted using its Public Key. In accordance to the S/MIME 3.1 standard, encrypted messages are encrypted using rsaEncryption. Messages are signed using rsaEncryption and SHA-1 hashing. Both encryption and signing of messages is done with the openssl CMS implementation CMS_encrypt and CMS_sign. The resulting message is called an IdP Reply.

Back at the Proxy, at step 4, the message will be decrypted using its Private Key. The values of the requested Attribute-Value Pairs are placed in a new URN. This message is once again encrypted in an S/MIME message, before being sent to the Client as a Proxy Reply.
Finally, in step 5 the client will decrypt the message and evaluate if it was received from the correct proxy, by checking the certificate.

Technical documentation

This chapter will describe the module in a technical mindset. We will discuss which dependencies will be required and how the program works using FunctionalC-diagrams[16] to clarify different parts in our story.

Dependencies

The following dependencies must be met for correctly compiling the module:

- FreeRADIUS 2.1.12
- OpenSSL 1.0.0-fips

[16] UML for the C programming language - ftp://ftp.software.ibm.com/common/ssi/rep_wh/n/RAW14058USEN/RAW14058USEN.PDF

Flow of module

In this chapter we will discuss different aspects of the module. For instance, we will describe the flow of the program and which functions are called to complete an action.

When FreeRADIUS is being started it will load all modules that are defined in sites-enabled and will create an instance where all the configuration is loaded that is defined in the radiusd.cnf or freeradius.conf. When our module is loaded, if it is defined in the sites-enabled directory, it will start the initialization of the module. During the initialization our module will read the certificates that must be defined in the RADIUS configuration files and will saved in the heap so that all other functions in our module will be able to access the certificates. The certificate that must be retained private is sometimes password-protected, for this reason some users must provide their password in the freeradius.conf or radiusd.cnf file. This is all described in step A1 and A2 in the flowchart (see the appendix).

The next step is to determine if the module should act as an IDP or proxy which will be done in step A3. In the case of a pre-proxy it will read the AVPs and look if the request is an acknowledgement or a request. When it is a request, the certificate in MIME-format will be injected in an AVP and will leave our module (this is done in B1 and B2). When the module has done its job all AVPs have leaved our module. FreeRADIUS will now send a message to the other side when all other modules have been exited successfully.

The RADIUS message has now been received by the IDP, if there is no loss in packages and everything is configured as it should be, and we will continue from step A4. The module is now in the post-auth section where it has received an ACCESS-ACCEPT or an ACCESS-REJECT in step A4, there is a possibility where the module receives an ACCESS-CHALLENGE in this case we assume that this doesn't happen, and will continue to step A5 or end the module in he case of an ACCESS-REJECT. All AVPs are now loaded and the module searches for an AVP named AVP_REQUEST_ATTRIBUTES in step A7. From now on the MIME-formatted messages that contain text will be unpacked and the URNs will be parsed in step C1 and C2. Also, the MIME-formatted messages that contains the certificates will be unpacked and from that moment the module will create a URN that holds the attributes in step C3, C4 and C5. Everything will be packed in an S/MIME-formatted message in step C6 that is encrypted with the public key that is unpacked in step C3 and will be inserted in an AVP in step C7.

When the next hop is made, the module will receive all data and unpack the S/MIME-formatted messages with the private key in step D1. The same is to be done with the S/MIME-formatted text which retrieves and parses the URN in step D2 and D3. The correct attributes needs to be searched in a VOMS-like service in step D4. When the attributes are found, an URN will be created that holds the attributes and is packed in an S/MIME-format message encrypted with the public key received from the client or service in step D5 and D6. This S/MIME-formatted message is then included inside an AVP in step D7.

File diagram

The FreeRadius module, counting only our created content, counts six source files, eight headers, and four structs. The source files are rlm_moonshot.c, mod_smime.c, request_handler_preproxy.c, proxymodule.c, idpmodule.c and x509_mod.c. There is one header per source file, in addition to a config.h and common.h header. The structs are attr_req_in, attr_req_out, avp_struct, and rlm_moonshot_t.

rlm_moonshot.c is the starting point of any exchange. When used, it will determine the path the module should take. This path is either from client to proxy, proxy to identity provider, or identity provider to proxy. rlm_moonshot.c also defines the avp_struct, attr_req_in and attr_req_out structs.

mod_smime.c is our module to handle any SMIME messages, with access to both packing and unpacking functionality. It is also able to add or strip headers and certificates of messages. request_handler_preproxy.c is the start of the path from client to proxy and proxy to identity provider. It determines if we are dealing with a request for authorization, or an already acknowledged authentication.

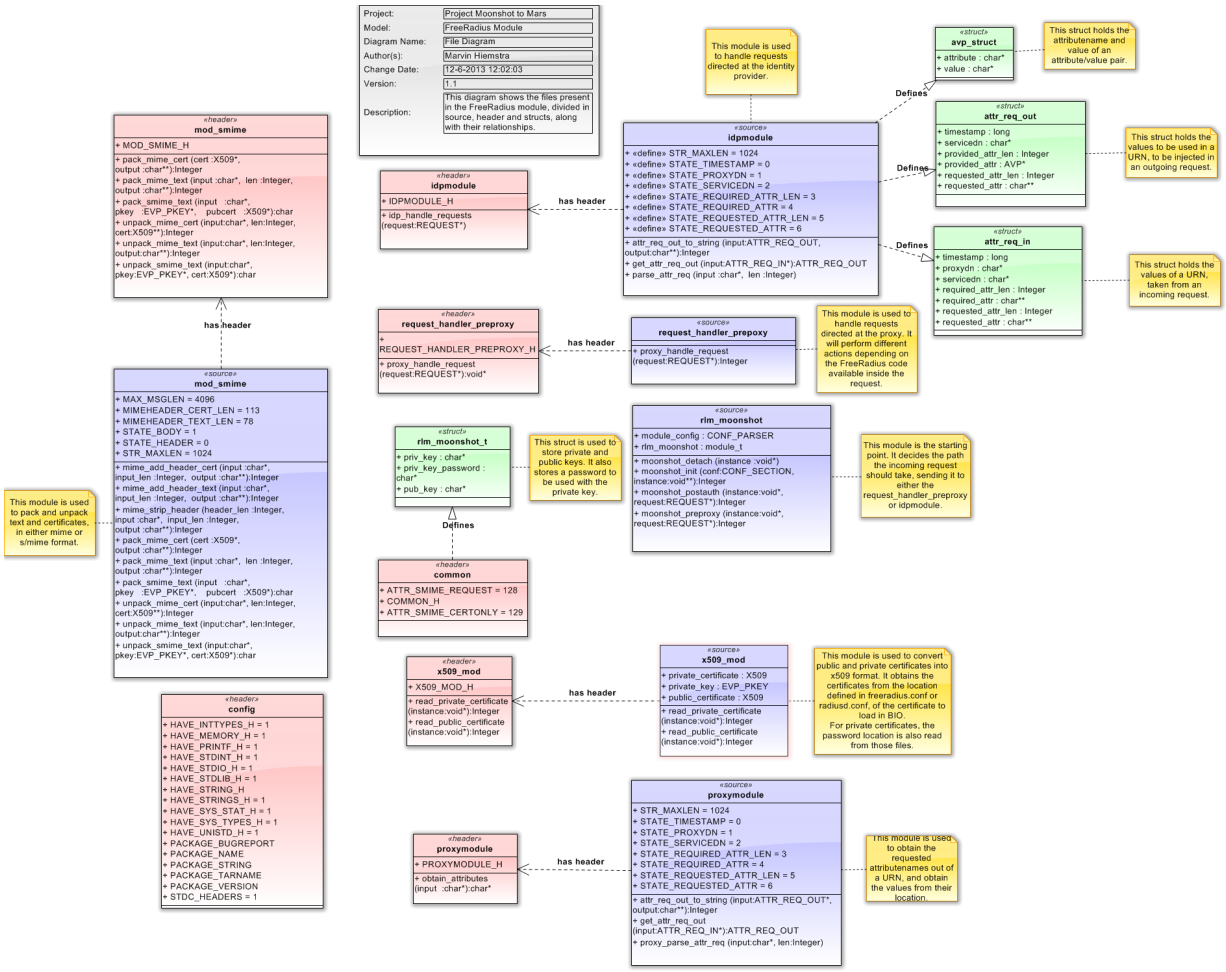
proxymodule.c and idpmodule.c are each other's counterparts, with roughly the same goals, adapted to their respective environment. The biggest difference is that in proxymodule.c, the requested attribute/value pairs are still unknown. Therefore it will extract these from the URN, request the obtained values from the VOMS server, and returns them. idpmodule.c is not concerned with this. Instead, it is able to find a certificate matching that of the proxy domain name.

x509_mod.c is the module we use when we work with x509 certificates. It is used to read public and private certificates. We also have a header called common.h that is included in all source files(except for mod_smime.c). This header is responsible for defining the rlm_moonshot_t struct. We also have config.h, which is used to store the settings. rlm_moonshot_t holds a private key, private key password, and public key. These are used for the encryption and decryption of the attribute/value pairs.

avp_struct contains two char pointers, representing an attributename and a value. attr_req_in and attr_req_out contain the data that is also present in our URN. They both contain a timestamp, required attribute/value pairs and how many of them, and the domain name of the requested service.

attr_req_in is used to store incoming requests. In addition to the above, it contains the domain name of the proxy, the names of the requested attribute/value pairs and how many of them we should expect.

attr_req_out stores the outgoing requests. It stores the provided attribute/values (as an avp_struct), and how many of them we have.



Known issues

General

- The URN and it's attributes is currently not encrypted.

idpmodule.c:

- The way the AVPs are reconstructed into a MIME message only properly allows for one client request.
- Finding matching certificates based on the provided DN and AVPs containing certificates is not yet functional.

request_handler_preproxy.c:

- The way the AVPs are reconstructed into a MIME message only properly allows for one client request.
- Seeing how not only pre_proxy, but also post_proxy hooks are used now, the files should be restructured to properly separate the pre_proxy and post_proxy functions. Currently, the post_proxy entry function also resides here.

mod_smime.c:

- unpack_smime_text is not yet functional. The OpenSSL CMS parser has trouble with large, memory-based BIOs which seems to be the reason why SMIME_read_CMS fails here.

Further development

The prove of concept only implements the RADIUS side of the specification. In order to create a proper pilot environment a client-side application should be developed alongside the RADIUS module.

The specification uses the SMIME standard to encrypt and decrypt messages containing authoritative statements from a third-party service (such as SurfConext) to a server application (such as the OpenSSH daemon). While this provides some form of protection against eavesdropping, the SMIME standard wasn't designed for this kind of traffic and in the used implementation, does not provide forward security.

The Diffie-Hellman Key Exchange method might be an interesting solution to this problem. By using the RADIUS Access-Challenge mechanism, it might be possible to complete a Diffie-Hellman Key Exchange between the client and the home institution RADIUS server.

Currently, our specification dictates that the server application using this service is the one requesting certain attributes about the authenticating user from third party services. The authenticating user should have more control over this decision and should at least be prompted with the fact that additional information is going to be requested elsewhere. A way to ensure that the client agreed to the requested attributes should be build in to the protocol specification.

Sources

(1)http://www.interlinknetworks.com/app_notes/eap-peap.htm

(2)https://tools.ietf.org/html/rfc5281 Chapter 9.1

(3) http://tools.ietf.org/html/draft-funk-eap-ttls-v1-01 Chapter 4.2

(4) http://docstore.mik.ua/orelly/networking_2ndEd/fire/ch14_08.htmWhole page

(5)http://tools.ietf.org/html/rfc2743 Chapter 5.3

(6)http://www.ssh.com/manuals/server-admin/44/User_Authentication_with_GSSAPI.htmlWhole page

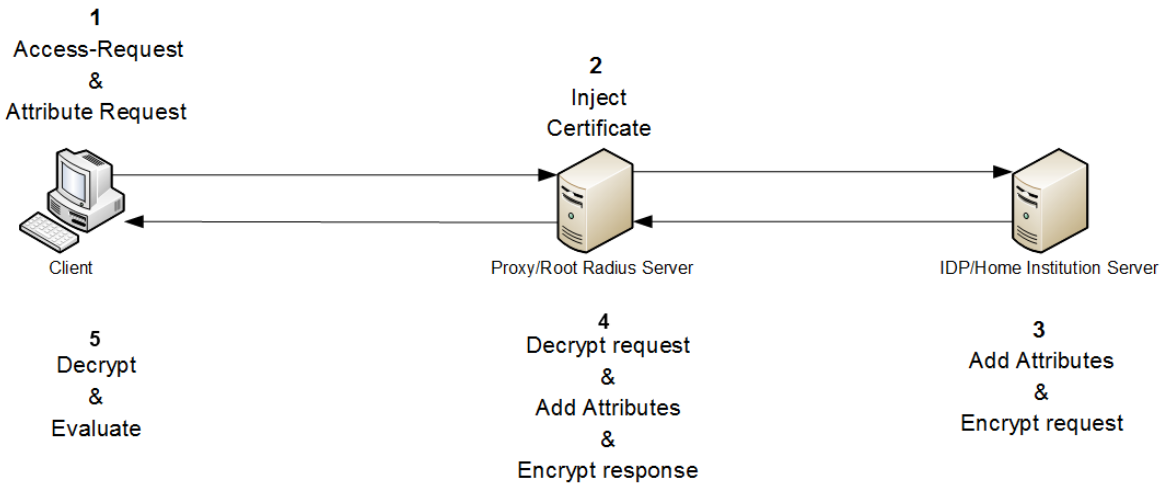
(7)http://www.ietf.org/rfc/rfc4251.txtIntroduction

(8)http://www.openssh.org/ Whole page

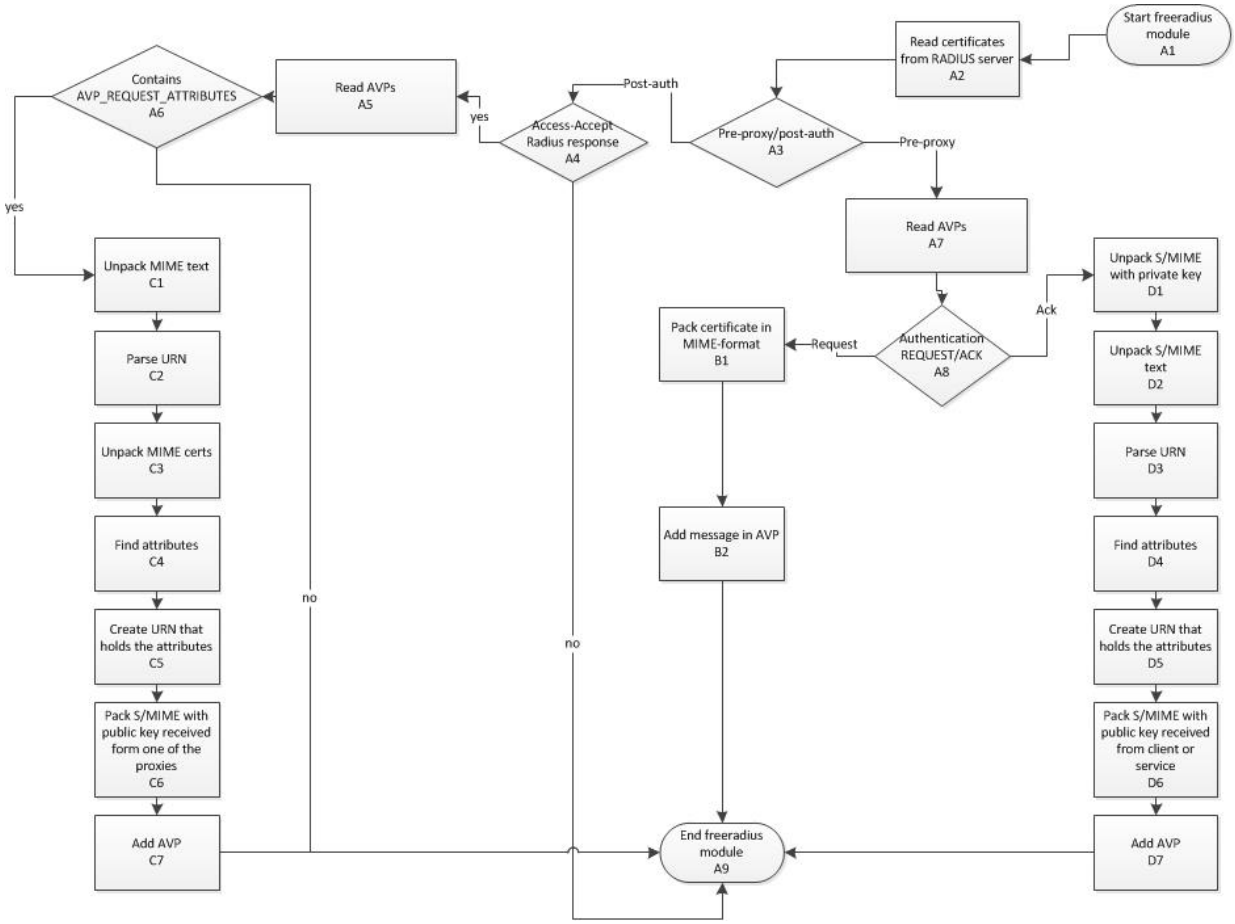
(9) http://www.untruth.org/~josh/security/radius/radius-auth.html Chapter 1 + 2

Appendix

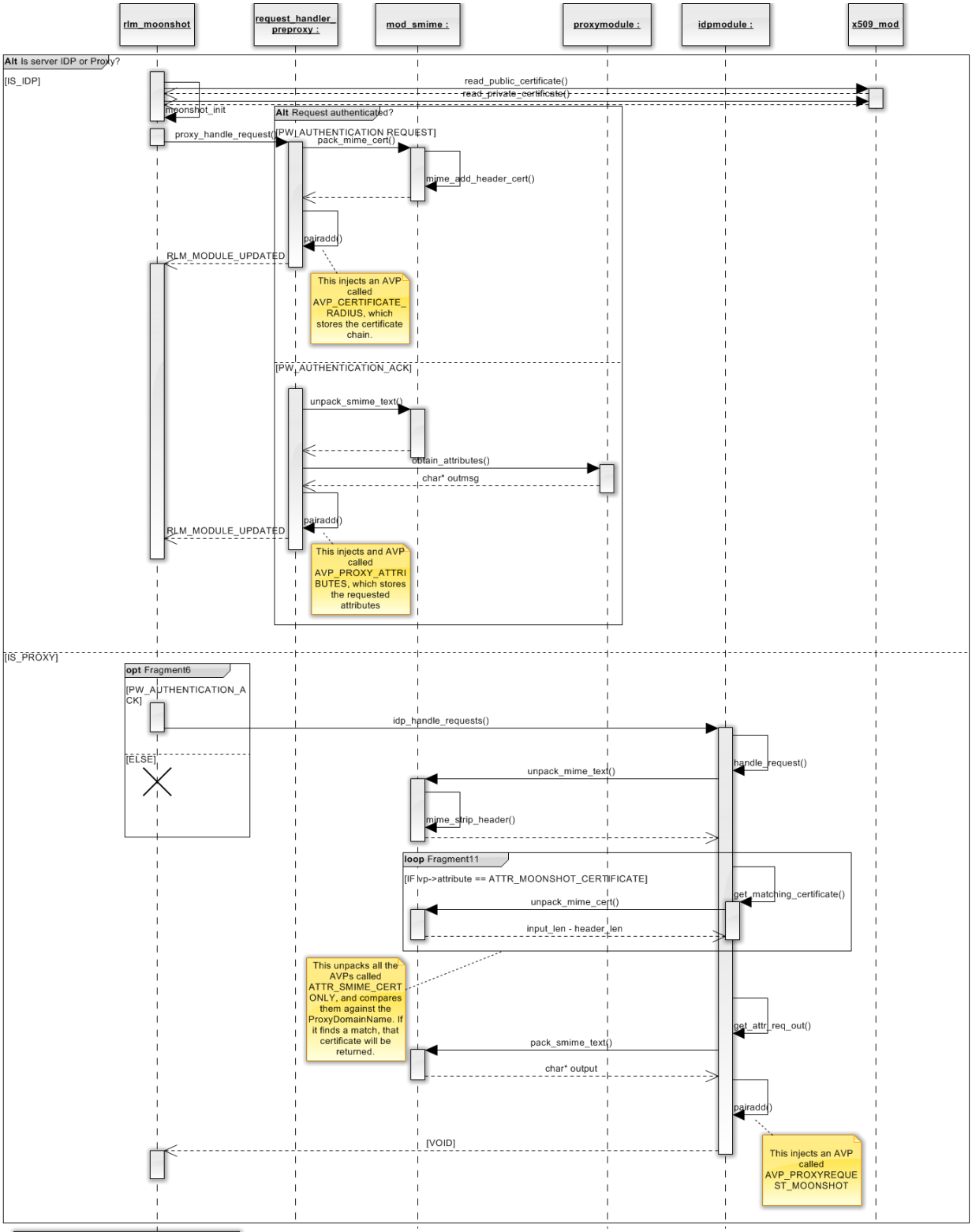
Topology diagram



Flowchart



Sequence Diagram



Project:	Project Moonshot to Mars
Model:	FreeRadius Module
Diagram Name:	Sequence Diagram
Author(s):	Marvin Hiemstra
Change Date:	12-6-2013 12:51:17
Version:	1.1
Description:	This diagram depicts the sequence of a request being handled, along with their calls and functions.

Source code

The entire source can be found at github.com/MoonshotNL/moonshotcode.git

rlm_moonshot.c

```
/*
This module is the starting point.
It decides the path the incoming request should take, sending it to either
the request_handler_preproxy or idpmodule.
*/
#include <freeradius-devel/ident.h>
RCSID("$Id$")

#include <freeradius-devel/radiusd.h>
#include <freeradius-devel/radius.h>
#include <freeradius-devel/modules.h>

#include "common.h"
#include "request_handler_preproxy.h"
#include "idpmodule.h"
#include "x509_mod.h"
#define AUTHENTICATION_REQUEST 1 //ACCEPT-REQUEST radius response
#define AUTHENTICATION_ACK 2  //ACCEPT-ACCEPT radius response

static const CONF_PARSER module_config[] = {
    { "pub_key",  PW_TYPE_STRING_PTR, offsetof(rlm_moonshot_t,pub_key),
NULL,  NULL}, //holds location of the public certificate
    { "priv_key",  PW_TYPE_STRING_PTR, offsetof(rlm_moonshot_t,priv_key),
NULL,  NULL}, //hols location of the private certificate
    { "priv_key_password",  PW_TYPE_STRING_PTR,
offsetof(rlm_moonshot_t,priv_key_password), NULL,  NULL}, //holds the
password of the private certificate
  { NULL, -1, 0, NULL, NULL }         /* end the list */
};

/* CONF_SECTION *conf seems to be the raw config data, rlm_moonshot_t *data
is our defined struct that will hold our data, CONF_PARSER module_config[]
are our config parser rules */
static int moonshot_init(CONF_SECTION *conf, void **instance)
{
        //Array that will store our parsed config data
        rlm_moonshot_t *data;

        data = rad_malloc(sizeof(rlm_moonshot_t));
        if (!data) {
                return -1;
        }
        memset(data, 0, sizeof(rlm_moonshot_t));

        //Parse the config file using conf, data and our parse rules in
module_config
        if (cf_section_parse(conf, data, module_config) < 0) {
                free(data); //rauwe data naar geformatte data
                return -1;
        }

        *instance = data;
```

```
    read_public_certificate(*instance);
    read_private_certificate(*instance);

        return 0;
}

/*
Handle pre-proxy requests, this is done by request_handler_preproxy.c
*/
static int moonshot_preproxy(void *instance, REQUEST *request)
{
        /* quiet the compiler */
        instance = instance;

    preproxy_handle_request(request);

        return RLM_MODULE_OK;
}

/*
Handle pre-proxy requests, this is done by request_handler_preproxy.c
*/
static int moonshot_postproxy(void *instance, REQUEST *request)
{
        /* quiet the compiler */
        instance = instance;

    postproxy_handle_request(request);

        return RLM_MODULE_OK;
}

/*
Gives the idp_module requests to handle, provided Radius gave us an
ACCESS_ACCEPT
*/
static int moonshot_postauth(void *instance, REQUEST *request)
{
        /* quiet the compiler */
        instance = instance;

        //Is it an Access-Accept and we're not a proxy?
        if (request->reply->code == PW_AUTHENTICATION_ACK && request-
>proxy_reply == NULL)
        {
                idp_handle_requests(request);
        }

        return RLM_MODULE_OK;
}

/*
Unregister our module to free up space
*/
static int moonshot_detach(void *instance)
{
        free(instance);
        return 0;
}

//Register our functions in the correct places
```

```
module_t rlm_moonshot = {
        RLM_MODULE_INIT,
        "moonshot",                     /* module name */
        RLM_TYPE_THREAD_SAFE,           /* type */
        moonshot_init,                  /* instantiation */
        moonshot_detach,                        /* detach */
        {
                NULL,                   /* authentication */
                NULL,                   /* authorization */
                NULL,                   /* preaccounting */
                NULL,                   /* accounting */
                NULL,                   /* checksimul */
                moonshot_preproxy,      /* pre-proxy */
                moonshot_postproxy,                     /* post-proxy */
                moonshot_postauth       /* post-auth */
        },
};
```

idpmodule.c

```
/*
This module is used to handle requests directed at the identity provider.
*/
#include <freeradius-devel/radiusd.h>
#include <freeradius-devel/radius.h>
#include <freeradius-devel/modules.h>
#include <freeradius-devel/libradius.h>

#include <openssl/x509.h>

#include "common.h"
#include "mod_smime.h"

#define STR_MAXLEN                                      1024

#define STATE_TIMESTAMP                                 0
#define STATE_PROXYDN                           1
#define STATE_SERVICEDN                                 2
#define STATE_REQUIRED_ATTR_LEN                 3
#define STATE_REQUIRED_ATTR                     4
#define STATE_REQUESTED_ATTR_LEN        5
#define STATE_REQUESTED_ATTR            6

extern EVP_PKEY *private_key;
extern X509             *public_certificate;

/*
This struct holds the attributename and value of an attribute/value pair.
*/
typedef struct avp_struct
{
        char *attribute;
        char *value;
} AVP;

/*
This struct holds the values of a URN, taken from an incoming request.
*/
typedef struct attr_req_in
{
        unsigned long timestamp;
        char *proxydn;
```

```
        char *servicedn;
        int required_attr_len;
        char **required_attr;
        int requested_attr_len;
        char **requested_attr;
} ATTR_REQ_IN;

/*
This struct holds the values to be used in a URN, to be injected in an
outgoing request.
*/
typedef struct attr_req_out
{
        unsigned long timestamp;
        char *servicedn;
        int provided_attr_len;
        AVP *provided_attr;
        int requested_attr_len;
        char **requested_attr;
} ATTR_REQ_OUT;

/*
This function reads a URN, and places it's information into a ATTR_REQ_IN
struct. It is dependant on the URN having the right structure, and knowing
the correct length of the URN.
*/
static ATTR_REQ_IN *parse_attr_req(char *input, int len)
{
   ATTR_REQ_IN *tmp_attr_req = rad_malloc(sizeof(ATTR_REQ_IN));
   int input_cur = 0;

   char item_tmp[STR_MAXLEN];
   int item_cur = 0;

   int attr_p = 0;

   int state = STATE_TIMESTAMP;

   while(input_cur <= len)
   {
      switch (state)
      {
         case STATE_TIMESTAMP:
            if (input[input_cur] == ':')
            {
               item_tmp[item_cur] = '\0';
               tmp_attr_req->timestamp = strtol(item_tmp, NULL, 10);
               state++;
               input_cur++;
               bzero(item_tmp, sizeof(char) * STR_MAXLEN);
               item_cur = 0;
               break;
            }
            item_tmp[item_cur] = input[input_cur];
            item_cur++;
            input_cur++;
            break;
         case STATE_PROXYDN:
            if (input[input_cur] == ':')
            {
               item_tmp[item_cur] = '\0';
```

```
                tmp_attr_req->proxydn = rad_malloc(sizeof(char) * (item_cur +
1));
                memcpy(tmp_attr_req->proxydn, item_tmp, sizeof(char) *
(item_cur + 1));
                state++;
                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
          case STATE_SERVICEDN:
             if (input[input_cur] == ':')
             {
                item_tmp[item_cur] = '\0';
                tmp_attr_req->servicedn = rad_malloc(sizeof(char) * (item_cur
+ 1));
                memcpy(tmp_attr_req->servicedn, item_tmp, sizeof(char) *
(item_cur + 1));
                state++;
                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
          case STATE_REQUIRED_ATTR_LEN:
             if (input[input_cur] == ':')
             {
                item_tmp[item_cur] = '\0';
                tmp_attr_req->required_attr_len = (int) strtol(item_tmp,
NULL, 10);

                if (tmp_attr_req->required_attr_len == 0)
                {
                   state += 2;
                }
                else
                {
                   state++;
                }

                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
          case STATE_REQUIRED_ATTR:
             if (input[input_cur] == ':')
             {
                item_tmp[item_cur] = '\0';
```

```
                if (attr_p == 0)
                {
                   tmp_attr_req->required_attr = rad_malloc(sizeof(char *));
                   tmp_attr_req->required_attr[attr_p] =
rad_malloc(sizeof(char) * (item_cur + 1));
                }
                else
                {
                   tmp_attr_req->required_attr = realloc(tmp_attr_req-
>required_attr, sizeof(char *) * (attr_p + 1));
                   tmp_attr_req->required_attr[attr_p] =
rad_malloc(sizeof(char) * (item_cur + 1));
                }

                memcpy(tmp_attr_req->required_attr[attr_p], item_tmp,
sizeof(char) * (item_cur + 1));
                attr_p++;

                if (attr_p >= tmp_attr_req->required_attr_len)
                {
                   state++;
                   attr_p = 0;
                }
                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
          case STATE_REQUESTED_ATTR_LEN:
             if (input[input_cur] == ':')
             {
                item_tmp[item_cur] = '\0';
                tmp_attr_req->requested_attr_len = (int) strtol(item_tmp,
NULL, 10);

                if (tmp_attr_req->required_attr_len == 0)
                {
                   state += 2;
                }
                else
                {
                   state++;
                }

                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
          case STATE_REQUESTED_ATTR:
             if (input_cur == len)
             {
```

```
                item_tmp[item_cur] = '\0';

                if (attr_p == 0)
                {
                   tmp_attr_req->requested_attr = rad_malloc(sizeof(char *));
                   tmp_attr_req->requested_attr[attr_p] =
rad_malloc(sizeof(char) * (item_cur + 1));
                }
                else
                {
                   tmp_attr_req->requested_attr = realloc(tmp_attr_req-
>required_attr, sizeof(char *) * (attr_p + 1));
                   tmp_attr_req->requested_attr[attr_p] =
rad_malloc(sizeof(char) * (item_cur + 1));
                }


                memcpy(tmp_attr_req->requested_attr[attr_p], item_tmp,
sizeof(char) * (item_cur + 1));
                attr_p++;

                if (attr_p >= tmp_attr_req->requested_attr_len)
                {
                   state++;
                   attr_p = 0;
                }
                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;
                break;
             }
             item_tmp[item_cur] = input[input_cur];
             item_cur++;
             input_cur++;
             break;
       }
   }
   return tmp_attr_req;
}

/*
Obtain the values of an attributevalue pair. !This is currently a dummy-
function!
*/
static AVP *get_avps_by_attributes(char **attributes, int length)
{
   //This function is to be implemented for the IDPs auathentication backend
   AVP *avp_list;
   int i;
   char *dummy_attribute = "DummyAttr";
   char *dummy_value = "DummyVal";

   avp_list = rad_malloc(sizeof(AVP) * length);
   if (!avp_list)
   {
      return NULL;
   }

   for (i = 0; i < length; i++)
   {
      avp_list[i].attribute = strdup(dummy_attribute);
```

```
      if (!avp_list[i].attribute)
         return NULL;

      avp_list[i].value = strdup(dummy_value);
      if (!avp_list[i].value)
         return NULL;
   }

   return avp_list;
}

/*
Transform an incoming request to an outgoing request.
*/
static ATTR_REQ_OUT *get_attr_req_out(ATTR_REQ_IN *input)
{
   ATTR_REQ_OUT *outstruct;
   AVP *pairs;

   outstruct = rad_malloc(sizeof(ATTR_REQ_OUT));
   memset(outstruct, 0, sizeof(ATTR_REQ_OUT));

   pairs = get_avps_by_attributes(input->required_attr, input-
>required_attr_len);
   if (!pairs)
   {
      return NULL;
   }

   outstruct->servicedn = input->servicedn;
   outstruct->provided_attr_len = input->required_attr_len;
   outstruct->provided_attr = pairs;
   outstruct->requested_attr_len = input->requested_attr_len;
   outstruct->requested_attr = input->requested_attr;
   outstruct->timestamp = (unsigned long) time(0);

   return outstruct;
}

/*
Reads the variables from an ATTR_REQ_OUT struct, and places it in a
correctly formatted URN.
*/
static int attr_req_out_to_string(ATTR_REQ_OUT *input, char **output)
{
   char buffer[STR_MAXLEN];
   int i;

   memset(buffer, 0, STR_MAXLEN);

   sprintf(buffer, "%ld:%s:%i:", input->timestamp, input->servicedn, input-
>provided_attr_len);
   for (i = 0; i < input->provided_attr_len; i++)
   {
      sprintf(buffer + strlen(buffer), "%s=%s:", input-
>provided_attr[i].attribute, input->provided_attr[i].value);
   }
   sprintf(buffer + strlen(buffer), "%i:", input->requested_attr_len);
   for (i = 0; i < input->requested_attr_len; i++)
   {
      if (i == input->requested_attr_len - 1)
```

```
            sprintf(buffer + strlen(buffer), "%s", input->requested_attr[i]);
        else
            sprintf(buffer + strlen(buffer), "%s:", input->requested_attr[i]);
    }

    *output = rad_malloc(strlen(buffer));
    strcpy(*output, buffer);
    return strlen(*output);
}

/*
Obtains an X509 certificate whose name matches the domainname present in the
request.
*/
static X509 *get_matching_certificate(REQUEST *request, char *dn)
{
        X509 *tmp_cert;
        char certmsg[4096];
        int cert_started = 0;

        memset(certmsg, 0, 4096);

        VALUE_PAIR *vp = request->packet->vps;
        do
        {
                if (cert_started)
                {
                        if (vp->attribute == ATTR_MOONSHOT_CERTIFICATE)
                        {
                                if (strncmp(certmsg, "Mime-Version 1.0",
strlen("Mime-Version 1.0") == 0))
                                {
                                        unpack_mime_cert(certmsg,
strlen(certmsg), &tmp_cert);
                                        if (strcmp(tmp_cert->name, dn) == 0)
                                        {
                                                return tmp_cert;
                                        }
                                        free(tmp_cert);
                                        memset(certmsg, 0, 4096);
                                }
                                strcat(certmsg, (char *)vp->data.octets);
                        }
                }
                else
                {
                        if (strncmp(certmsg, "Mime-Version 1.0",
strlen("Mime-Version 1.0") == 0))
                        {
                                strcat(certmsg, (char *)vp->data.octets);
                                cert_started = 1;
                        }
                }
        } while ((vp = vp->next) != 0);

        unpack_mime_cert(certmsg, strlen(certmsg), &tmp_cert);

        if (strcmp(tmp_cert->name, dn) == 0)
        {
                return tmp_cert;
        }
```

```
        free(tmp_cert);
        return NULL;
}

/*
Determine the length of a URN and parse it's values. The resulting string is
divided in parts of 250 characters each to adhere to FreeRadius' policy, and
then inserted into the request as an attributevalue pair called
Moonshot_Request.
*/
static void handle_request(REQUEST *request, char *raw_input)
{
        char *input_data;
        int input_len;
        char *output_data;
        int output_len;
        char *smime_msg;
        char substr[250];
        int i;
        int msg_len;
        ATTR_REQ_OUT *outstruct;
        VALUE_PAIR *avp_smime;

        input_len = unpack_mime_text(raw_input, strlen(raw_input),
&input_data);
        ATTR_REQ_IN *attr_request = parse_attr_req(input_data, input_len);
        if (!attr_request)
        {
                return;
        }

        //X509 *cert = get_matching_certificate(request, attr_request-
>proxydn);
        //if (!cert)
        //{
        //      return;
        //}

        outstruct = get_attr_req_out(attr_request);
        output_len = attr_req_out_to_string(outstruct, &output_data);
        pack_mime_text(output_data, strlen(output_data), &smime_msg);
        for (i = 0; i <= (strlen(smime_msg) / 250); i++)
        {
                msg_len = i == (strlen(smime_msg) / 250) ? strlen(smime_msg)
% 250 : 250;
                memcpy(substr, &smime_msg[i * 250], msg_len);
                substr[msg_len] = '\0';
                avp_smime = pairmake("Moonshot-IDPReply", substr, T_OP_EQ);
                pairadd(&request->reply->vps, avp_smime);
        }
        return;
}

/*
Read a request to determine if it contains any attributes called
ATTR_MOONSHOT_REQUEST. The value of this attribute should be a URN. The
request along with the valuepair is then handled by handle_request()
*/
void idp_handle_requests(REQUEST *request)
{
```

```
        VALUE_PAIR *vp = request->packet->vps;
        int found = 0;
        char message[4096];
        memset(message, 0, 4096);
        do
        {
                if (vp->attribute == ATTR_MOONSHOT_REQUEST)
                {
                        found = 1;
                        strcat(message, (char *)vp->data.octets);
                }
        } while ((vp = vp->next) != 0);
        if (found)
        {
                handle_request(request, message);
        }
}
```

request_handler_preproxy.c

```
/*
This module is used to handle requests directed at the proxy.
It will perform different actions depending on the FreeRadius code available
inside the request.
*/

#include <freeradius-devel/radiusd.h>
#include <freeradius-devel/radius.h>
#include <freeradius-devel/modules.h>

#include "common.h"
#include "mod_smime.h"
#include "x509_mod.h"
#include "proxymodule.h"

extern X509 *public_certificate;
extern X509 *private_certificate;
extern EVP_PKEY *private_key;

/*
Handles requests before they have been handled by the proxy-server.
*/
int preproxy_handle_request(REQUEST *request)
{
        char message[4096];
        int found = 0;
        int i;
        int avp_msglen = 0;
        char *cert_message;
        char substr[251];
        VALUE_PAIR *vp;
        switch (request->packet->code) //it's allowed to handle multiple
requests, the request type is based on radius responses
        {
        case PW_AUTHENTICATION_REQUEST:
                pack_mime_cert(public_certificate, &cert_message);
                VALUE_PAIR *avp_certificate;

                for (i = 0; i <= (strlen(cert_message) / 250); i++)
                {
                        avp_msglen = i == (strlen(cert_message) / 250) ?
strlen(cert_message) % 250 : 250;
```

```
                        memcpy(substr, &cert_message[i * 250], avp_msglen);
                        substr[avp_msglen] = '\0';
                        avp_certificate = pairmake("Moonshot-Certificate",
substr, T_OP_EQ);
                        pairadd(&request->proxy->vps, avp_certificate); //add
AVP
                }
                //avp_certificate = pairmake("Moonshot-Certificate",
cert_message, T_OP_EQ); //AVP_CERTIFICATE_RADIUS is an AVP that stores the
certificate chain
                //pairadd(&request->reply->vps, avp_certificate); //add AVP
                return RLM_MODULE_UPDATED;                      //we are
basically saying that our AVPs are updated
        }
}

/*
Handles request that have been handled by a proxy, and thus have a Radius-
reply
*/
int postproxy_handle_request(REQUEST *request)
{
        char message[4096];
        int found = 0;
        int i;
        int avp_msglen = 0;
        char *cert_message;
        char substr[251];
        VALUE_PAIR *vp, *vp_prev;
        char *message_attributes, *out_urn, *out_message;

        switch (request->proxy_reply->code)
        {
                case PW_AUTHENTICATION_ACK: //If we're passing through an
ACCESS-ACCEPT
                        memset(message, 0, 4096);

                        vp = request->proxy_reply->vps;
                        do {
                                if (vp->attribute == ATTR_MOONSHOT_IDPREPLY)
                                {
                                        found = 1;
                                        strncat(message, vp->data.octets, vp-
>length);
                                        if (vp_prev != NULL)
                                        {
                                                vp_prev->next = vp->next;
                                                free(vp);
                                                vp = vp_prev;
                                        }
                                }
                                vp_prev = vp;
                        } while ((vp = vp -> next) != 0);

                        if (found)
                        {
                                unpack_mime_text(message, strlen(message),
&message_attributes);
                                out_urn =
obtain_attributes(message_attributes);
```

```
                                pack_mime_text(out_urn, strlen(out_urn),
&out_message);

                                VALUE_PAIR *avp_attributes;

                                for (i = 0; i <= (strlen(out_message) / 250);
i++)
                                {
                                        avp_msglen = i == (strlen(out_message)
/ 250) ? strlen(out_message) % 250 : 250;
                                        memcpy(substr, &out_message[i * 250],
avp_msglen);
                                        substr[avp_msglen] = '\0';
                                        avp_attributes = pairmake("Moonshot-
ProxyReply", substr, T_OP_EQ);
                                        pairadd(&request->proxy_reply->vps,
avp_attributes);
                                }
                                return RLM_MODULE_UPDATED;
                        }
        }
        return NULL;
}
```

mod_smime.c

```
/*
This module is used to pack and unpack text and certificates, in either mime
or s/mime format.
*/
#include "mod_base64.h"
#include <openssl/pem.h>
#include <openssl/cms.h>
#include <openssl/err.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define STR_MAXLEN                              1024
#define MIMEHEADER_TEXT_LEN             78
#define MIMEHEADER_CERT_LEN             113

#define STATE_HEADER   0
#define STATE_BODY             1

#define MAX_MSGLEN     4096

/*
Return a substring of the input, up until (but not including) the null
terminator.
*/
static void remove_nl(char **string)
{
        char *buffer;
        int i, buf_cur;

        buffer = calloc(strlen(*string) + 1, sizeof(char));
        buf_cur = 0;

        for (i = 0; i < strlen(*string); i++)
        {
                if ((*string)[i] != '\n')
```

```
                {
                        buffer[buf_cur] = (*string)[i];
                        buf_cur++;
                }
        }

        free(*string);
        *string = buffer;
}

/*
This function will strip the header off a mime-message, so it can be read
"normally"
*/
static int mime_strip_header(int header_len, char *input, int input_len,
char **output)
{
        *output = calloc(1, input_len - header_len);
        memcpy(*output, input + header_len, input_len - header_len);
        return input_len - header_len;
}

/*
This function will add a mime-header to your input. This header will define
the content of your input as base64 encoded text.
If you are planning to add a header to a certificate(chain), see
mime_add_header_cert();
*/
static int mime_add_header_text(char *input, int input_len, char **output)
{
        char *header = "Mime-version: 1.0\nContent-Type: text/plain\nContent-
Transfer-Encoding: base64\n\n";
        *output = malloc((sizeof(char) * input_len) + (sizeof(char) *
MIMEHEADER_TEXT_LEN) + 1);
        strcpy(*output, header);
        strcat(*output, input);
        return input_len + MIMEHEADER_TEXT_LEN + 1;
}

/*
This function will add a mime-header to your input. This header will define
the content of your input as a base64 encoded certificate(chain).
If you are planning to add a header to regular text instead, see
mime_add_header_text();
*/
static int mime_add_header_cert(char *input, int input_len, char **output)
{
        char *header = "Mime-Version: 1.0\nContent-Type: application/pkcs7-
mime; smime-type=certs-only\nContent-Transfer-Encoding: base64\n\n";
        *output = malloc(input_len + MIMEHEADER_CERT_LEN + 1);
        strcpy(*output, header);
        strcat(*output, input);
        return input_len + MIMEHEADER_CERT_LEN + 1;
}

/*
This function will encode the input in base64 format, and return a mime-
message. This input shall be treated as regular text.
This will automatically call add_mime_header_text(). This should not be done
manually.
To pack the input as an s/mime-message, see pack_smime_text();
```

```
*/
int pack_mime_text(char *input, int len, char **output)
{
        int out_len = 0;
        char *base64_input;
        base64_input = base64(input, len);

        out_len = mime_add_header_text(base64_input, strlen(base64_input),
output);

        return out_len;
}

/*
This function is used to return a byte-array from a mime-message input.
The message gets it's header stripped automatically. It is assumed the
content of the message (minus the header) are base64 encoded.
*/
int unpack_mime_text(char *input, int len, char **output)
{
        char *base64_out;
        int base64_len;

        base64_len = mime_strip_header(MIMEHEADER_TEXT_LEN, input, len,
&base64_out);
        remove_nl(&base64_out);
        *output = unbase64(base64_out, strlen(base64_out));
        return strlen(*output);
}

/*
This function will encode the input in base64 format, and return a mime-
message. This input shall be treated as a certificate(chain).
This function will automatically call add_mime_header_text(). This should
not be done manually.
*/
int pack_mime_cert(X509 *cert, char **output)
{
        BIO *bio = NULL;
        char *outbuffer;
        BUF_MEM *bptr;

        outbuffer = malloc(5120);
        memset(outbuffer, 0, 5120);

        bio = BIO_new(BIO_s_mem());
        if (!bio)
        {
                return -1;
        }

        if (!PEM_write_bio_X509(bio, cert))
        {
                BIO_free(bio);
                return -1;
        }

        BIO_get_mem_ptr(bio, &bptr);
        outbuffer = strndup(bptr->data, bptr->length);

        mime_add_header_cert(outbuffer, strnlen(outbuffer, 5120), output);
```

```
        free(outbuffer);
        return 0;
}

/*
This function unpacks a certificate(chain) from a mime-message, and return
an integer indicating it's success.
The message gets it's header stripped automatically. It is assumed the
content of the message (minus the header) are base64 encoded.
*/
int unpack_mime_cert(char *input, int len, X509 **cert)
{
        *cert = NULL;
        BIO *bio = NULL;
        char *noheader;

        mime_strip_header(MIMEHEADER_CERT_LEN, input, strlen(input),
&noheader);

        bio = BIO_new_mem_buf(noheader, -1);
        if (!bio)
        {
                return -1;
        }

        PEM_read_bio_X509(bio, cert, 0, NULL);
        BIO_free(bio);
        if (!*cert)
        {
                return -1;
        }

        return 0;
}

/*
This function will encode the input in base64 format, and return a s/mime-
message. This input shall be treated as regular text.
To pack the input as a regular mime-message, see pack_mime_text();
The content will be encrypted using the private key and public certificate
supplemented.
*/
char *pack_smime_text(char *input, EVP_PKEY *pkey, X509 *pubcert)
{
   STACK_OF(X509) *recips = NULL;
   CMS_ContentInfo *cms_sig = NULL, *cms_enc = NULL;
   BIO *bio_in = NULL, *bio_sig = NULL, *bio_out = NULL;
   BUF_MEM *bptr;
   char *output = NULL;
   int flags = CMS_STREAM;

   OpenSSL_add_all_algorithms();

   recips = sk_X509_new_null();
   if (!recips || !sk_X509_push(recips, pubcert))
   {
      printf("recips || sk_X509_push error\n");
      exit(1);
   }

   bio_in = BIO_new_mem_buf(input, -1);
```

```
   bio_sig = BIO_new(BIO_s_mem());
   bio_out = BIO_new(BIO_s_mem());

   if (!bio_in || !bio_sig || !bio_out)
   {
      printf("bio_in || bio_sig || bio_out error\n");
      exit(1);
   }

   cms_sig = CMS_sign(pubcert, pkey, NULL, bio_in, CMS_DETACHED|CMS_STREAM);
   if (!cms_sig)
   {
      printf("cms_sig error\n");
      exit(1);
   }

   if (!SMIME_write_CMS(bio_sig, cms_sig, bio_in, CMS_DETACHED|CMS_STREAM))
   {
      printf("Error SMIME_write_CMS bio_sig");
      exit(1);
   }

   cms_enc = CMS_encrypt(recips, bio_sig, EVP_des_ede3_cbc(), flags);

   if (!cms_enc)
   {
      printf("cms error\n");
      exit(1);
   }

   if (!SMIME_write_CMS(bio_out, cms_enc, bio_sig, flags))
   {
      printf("SMIME write error\n");
      exit(1);
   }

   BIO_get_mem_ptr(bio_out, &bptr);
   output = bptr->data;
   output = strndup(bptr->data, bptr->length);

   CMS_ContentInfo_free(cms_sig);
   CMS_ContentInfo_free(cms_enc);
   BIO_free(bio_in);
   BIO_free(bio_sig);
   BIO_free(bio_out);

   return output;
}

/*
This function will unpack an s/mime message and return it as a char pointer.
This needs both the correct private key and x509 certificate.
*/
char *unpack_smime_text(char *input, EVP_PKEY *pkey, X509 *cert)
{
        BIO *bio_in = NULL, *bio_out = NULL;
        CMS_ContentInfo *cms = NULL;
        char *output = NULL;
        BUF_MEM *bptr = NULL;

        OpenSSL_add_all_algorithms();
```

```
        bio_in = BIO_new_mem_buf(input, -1);
        bio_out = BIO_new(BIO_s_mem());

        if (!bio_in || !bio_out)
        {
                DEBUG("dectext: error creating bio_in, bio_dec or bio_out");
                exit(1);
        }
        DEBUG("About to read the CMS");
        cms = SMIME_read_CMS(bio_in, NULL);
        if (!cms)
        {
                DEBUG("Error parsing message to CMS");
                exit(1);
        }

        DEBUG("About to CMS_decrypt");
        if (!CMS_decrypt(cms, pkey, cert, NULL, bio_out, 0))
        {
                DEBUG("Error decrypting message");
                exit(1);
        }

        BIO_get_mem_ptr(bio_out, &bptr);
        output = strndup(bptr->data, bptr->length);

        CMS_ContentInfo_free(cms);
    BIO_free(bio_in);
    BIO_free(bio_out);

        return output;
}
```

proxymodule.c

```
/*
This module is used to obtain the requested attributenames out of a URN, and
obtain the values from their location.*/
#include <freeradius-devel/radiusd.h>
#include <freeradius-devel/radius.h>
#include <freeradius-devel/modules.h>
#include <freeradius-devel/libradius.h>

#include "common.h"
#include "mod_smime.h"

#define STR_MAXLEN                                      1024

#define STATE_TIMESTAMP                                 0
#define STATE_SERVICEDN                                 1
#define STATE_PROVIDED_ATTR_LEN                 2
#define STATE_PROVIDED_ATTR                     3
#define STATE_REQUESTED_ATTR_LEN        4
#define STATE_REQUESTED_ATTR            5
#define STATE_LIM                                       5

typedef struct avp_struct
{
        char *attribute;
        char *value;
} AVP;
```

```
/*
This struct holds the values of a URN, taken from an incoming request.
*/
typedef struct attr_req_in
{
        unsigned long timestamp;
        char *servicedn;
        int provided_attr_len;
        AVP *provided_attr;
        int requested_attr_len;
        char **requested_attr;
} ATTR_REQ_IN;

/*
This struct holds the values to be used in a URN, to be injected in an
outgoing request.
*/
typedef struct attr_req_out
{
        unsigned long timestamp;
        char *servicedn;
        int requested_attr_len;
        AVP *requested_attr;
} ATTR_REQ_OUT;

/*
Transform the input to an attributevalue pair
*/
AVP *atoavp(char *input)
{
        AVP *tmp_avp;
        int sep_offset = 0;

        while (sep_offset < strlen(input) && input[sep_offset] != '=')
                sep_offset++;

        if (sep_offset == strlen(input) - 1)
                return NULL;

        tmp_avp = rad_malloc(sizeof(AVP));
        tmp_avp->attribute = strndup(input, sep_offset);
        tmp_avp->value = strdup(input + sep_offset + 1);

        return tmp_avp;
}

/*
This function reads a URN, and places it's information into a ATTR_REQ_IN
struct. It is dependant on the URN having the right structure, and knowing
the correct length of the URN.
*/
ATTR_REQ_IN *proxy_parse_attr_req(char *input, int len)
{
        ATTR_REQ_IN *tmp_attr_req = rad_malloc(sizeof(ATTR_REQ_IN));
        AVP *tmp_avp;

        int input_cur = 0;
        int attr_p = 0;
        int item_cur = 0;
```

```
        char item_tmp[STR_MAXLEN];

        int state = STATE_TIMESTAMP;

        while(input_cur < len && state <= STATE_LIM)
        {
                switch (state)
                {
                        case STATE_TIMESTAMP:
                                if (input[input_cur] == ':')
                                {
                                        item_tmp[item_cur] = '\0';
                                        tmp_attr_req->timestamp =
strtol(item_tmp, NULL, 10);
                                        state++;
                                        input_cur++;
                                        bzero(item_tmp, sizeof(char) *
STR_MAXLEN);
                                        item_cur = 0;
                                        break;
                                }
                                item_tmp[item_cur] = input[input_cur];
                                item_cur++;
                                input_cur++;
                                break;
             case STATE_SERVICEDN:
                                if (input[input_cur] == ':')
                                {
                                        item_tmp[item_cur] = '\0';
                                        tmp_attr_req->servicedn =
rad_malloc(sizeof(char) * (item_cur + 1));
                                        memcpy(tmp_attr_req->servicedn,
item_tmp, sizeof(char) * (item_cur + 1));
                                        state++;
                                        input_cur++;
                                        bzero(item_tmp, sizeof(char) *
STR_MAXLEN);
                                        item_cur = 0;
                                        break;
                                }
                                item_tmp[item_cur] = input[input_cur];
                                item_cur++;
                                input_cur++;
                                break;
                        case STATE_PROVIDED_ATTR_LEN:
                                if (input[input_cur] == ':')
                                {
                                        item_tmp[item_cur] = '\0';
                                        tmp_attr_req->provided_attr_len =
(int) strtol(item_tmp, NULL, 10);
                                        tmp_attr_req->provided_attr =
rad_malloc(sizeof(AVP) * tmp_attr_req->provided_attr_len);

                                        if (tmp_attr_req->provided_attr_len ==
0)
                                        {
                                                state += 2;
                                        }
                                        else
                                        {
                                                state++;
```

```
                                        }

                                        input_cur++;
                                        bzero(item_tmp, sizeof(char) *
STR_MAXLEN);
                                        item_cur = 0;
                                        break;
                                }
                                item_tmp[item_cur] = input[input_cur];
                                item_cur++;
                                input_cur++;
                                break;
                        case STATE_PROVIDED_ATTR:
                                if (input[input_cur] == ':')
                                {
                                        item_tmp[item_cur] = '\0';

                                        tmp_avp = atoavp(item_tmp);
                                        memcpy(tmp_attr_req->provided_attr +
(attr_p * sizeof(AVP)), tmp_avp, sizeof(AVP));
                                        free(tmp_avp);
                                        attr_p++;

                                        if (attr_p >= tmp_attr_req-
>provided_attr_len)
                                        {
                                                state++;
                                                attr_p = 0;
                                        }
                                        input_cur++;
                                        bzero(item_tmp, sizeof(char) *
STR_MAXLEN);
                                        item_cur = 0;
                                        break;
                                }
                                item_tmp[item_cur] = input[input_cur];
                                item_cur++;
                                input_cur++;
                                break;
                case STATE_REQUESTED_ATTR_LEN:
                        if (input[input_cur] == ':')
                        {
                                item_tmp[item_cur] = '\0';
                                tmp_attr_req->requested_attr_len = (int)
strtol(item_tmp, NULL, 10);

                                if (tmp_attr_req->requested_attr_len == 0)
                                {
                                        state += 2;
                                }
                                else
                                {
                                        state++;
                                }

                                input_cur++;
                                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                                item_cur = 0;
                                break;
                        }
                        item_tmp[item_cur] = input[input_cur];
```

```
                        item_cur++;
                        input_cur++;
                        break;
                case STATE_REQUESTED_ATTR:
                        if (input[input_cur] == ':')
                        {
                                item_tmp[item_cur] = '\0';

                                if (attr_p == 0)
                                {
                                        tmp_attr_req->requested_attr =
rad_malloc(sizeof(char *));
                                        tmp_attr_req->requested_attr[attr_p] =
rad_malloc(item_cur + 1);
                                }
                                else
                                {
                                        tmp_attr_req->requested_attr =
realloc(tmp_attr_req->requested_attr, sizeof(char *) * (attr_p + 1));
                                        tmp_attr_req->requested_attr[attr_p] =
rad_malloc(item_cur + 1);
                                }

                                memcpy(tmp_attr_req->requested_attr[attr_p],
item_tmp, item_cur + 1);
                                attr_p++;

                                if (attr_p >= tmp_attr_req-
>requested_attr_len)
                                {
                                        state++;
                                        attr_p = 0;
                                }
                                input_cur++;
                                bzero(item_tmp, STR_MAXLEN);
                                item_cur = 0;
                                break;
                        }
                        item_tmp[item_cur] = input[input_cur];
                        item_cur++;
                        input_cur++;
                        break;
                }
        }
        return tmp_attr_req;
}

/*
Obtain the values of an attributevalue pair. !This is currently a dummy-
function!
*/
static AVP *get_avps_by_attributes(AVP *attributes, int length)
{
   //This function is to be implemented for the IDPs auathentication backend
   AVP *avp_list;
   int i;
   char *dummy_attribute = "DummyAttr";
   char *dummy_value = "DummyVal";

   avp_list = rad_malloc(sizeof(AVP) * length);
   if (!avp_list)
```

```
   {
      return NULL;
   }

   for (i = 0; i < length; i++)
   {
      avp_list[i].attribute = strdup(dummy_attribute);
      if (!avp_list[i].attribute)
         return NULL;

      avp_list[i].value = strdup(dummy_value);
      if (!avp_list[i].value)
         return NULL;
   }

   return avp_list;
}

/*
Transform an incoming request to an outgoing request.
*/
ATTR_REQ_OUT *get_attr_req_out(ATTR_REQ_IN *input)
{
        ATTR_REQ_OUT *outstruct;
        AVP *pairs;

        outstruct = rad_malloc(sizeof(ATTR_REQ_OUT));
        memset(outstruct, 0, sizeof(ATTR_REQ_OUT));

        pairs = get_avps_by_attributes(input->provided_attr, input-
>provided_attr_len);
        if (!pairs)
        {
                return NULL;
        }

        outstruct->timestamp = (long) time(0);
        outstruct->servicedn = input->servicedn;
        outstruct->requested_attr_len = input->requested_attr_len;
        outstruct->requested_attr = pairs;

        return outstruct;
}

/*
Reads the variables from an ATTR_REQ_OUT struct, and places it in a
correctly formatted URN.
*/
int attr_req_out_to_string(ATTR_REQ_OUT *input, char **output)
{
        char buffer[STR_MAXLEN];
        int i;

        memset(buffer, 0, STR_MAXLEN);

        sprintf(buffer, "%lu:%s:%i:", input->timestamp, input->servicedn,
input->requested_attr_len);
        for (i = 0; i < input->requested_attr_len; i++)
        {
                if (i == input->requested_attr_len - 1)
                {
```

```
                        sprintf(buffer + strlen(buffer), "%s=%s", input-
>requested_attr[i].attribute, input->requested_attr[i].value);
                }
                else
                {
                        sprintf(buffer + strlen(buffer), "%s=%s:", input-
>requested_attr[i].attribute, input->requested_attr[i].value);
                }
        }
        *output = rad_malloc(strlen(buffer));
        strcpy(*output, buffer);
        return strlen(*output);
}

/*
Extract the attributes from an charpointer
*/
char *obtain_attributes(char *input)
{
        ATTR_REQ_IN *instruct;
        ATTR_REQ_OUT *outstruct;
        char *outmsg;

        instruct = proxy_parse_attr_req(input, strlen(input));
        outstruct = get_attr_req_out(instruct);
        attr_req_out_to_string(outstruct, &outmsg);

        return outmsg;
}
```

x509_mod.c

```
/*
This module is used to convert public and private certificates into x509
format.
It obtains the certificates from the location defined in freeradius.conf or
radiusd.conf, of the certificate to load in BIO.
For private certificates, the password location is also read from those
files.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/pem.h>
#include "common.h"

X509 *public_certificate;
X509 *private_certificate;
EVP_PKEY *private_key;

/*
 read_public_certificate

 Use this function to get the public certificate in X509-format.
 This function uses the location, that is defined in the freeradius.conf or
radiusd.conf, of the certificate to load in BIO.
 In the next step it will convert it in a X509-format and returns the memory
location of the X509-certificate.
 */

int read_public_certificate(void *instance)
{
    BIO *tbio = NULL;
    public_certificate = calloc(1, sizeof(X509));
    char *cert;
    rlm_moonshot_t *data;

    data = (rlm_moonshot_t *)instance;
    cert = data->pub_key;                    //get the location of the public
certificate that is defined in the configuration files
    tbio = BIO_new_file(cert, "r");

    public_certificate = PEM_read_bio_X509(tbio, NULL, 0, NULL);

    if(!public_certificate)
    {
        return -1;
    }

    return 0;
}

/*
 read_private_certificate

 Use this function to get the private certificate in X509-format.
 This function uses the location, that is defined in the freeradius.conf or
radiusd.conf, of the certificate to load in BIO.
 In the next step it will convert it in a X509-format and returns the memory
location of the X509-certificate.
```

```
 Some certificates are secured by a password, therefore it reads the
password from the freeradius.conf or radiusd.cnf file.
 The password has to be defined in this configuration files for correctly
reading and returning the certificate in X509-format.


 */

int read_private_certificate(void *instance)
{
        BIO *tbio = NULL;
        private_certificate = calloc(1, sizeof(X509));
        char *cert;
        char *password;
        rlm_moonshot_t *data;
        int size;

        data = (rlm_moonshot_t *)instance;
        cert = data->priv_key;
        password = data->priv_key_password;         //get the password of the
private key that is defined in the configuration files

        tbio = BIO_new_file(cert, "r");

        private_certificate = PEM_read_bio_X509(tbio, NULL, NULL, password);
        private_key = PEM_read_bio_PrivateKey(tbio, NULL, 0, password);
        if(!private_certificate || !private_key)
        {
                return -1;
        }

        return 0;
}
```

mod_base64.c

```
/*
This module is used to format strings to Base64 and vice versa.
*/
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <string.h>

/*
Format a string in Base64.
*/
char *base64(const unsigned char *input, int length)
{
        BIO *bmem, *b64;
        BUF_MEM *bptr;
        char *buff;

        b64 = BIO_new(BIO_f_base64());
        bmem = BIO_new(BIO_s_mem());
        b64 = BIO_push(b64, bmem);
        BIO_write(b64, input, length);
        BIO_flush(b64);
        BIO_get_mem_ptr(b64, &bptr);

        buff = (char *)malloc(bptr->length);
        memcpy(buff, bptr->data, bptr->length-1);
        buff[bptr->length-1] = 0;

        BIO_free_all(b64);

        return buff;
}

/*
Change Base64 formattedtext to a string.
*/
char *unbase64(unsigned char *input, int length)
{
        BIO *b64, *bmem;

        char *buffer = (char *)malloc(length);
        memset(buffer, 0, length);

        b64 = BIO_new(BIO_f_base64());
        bmem = BIO_new_mem_buf(input, length);
        bmem = BIO_push(b64, bmem);
        BIO_set_flags(bmem, BIO_FLAGS_BASE64_NO_NL);

        BIO_read(bmem, buffer, length);

        BIO_free_all(bmem);

        return buffer;
}
```

include/common.h

```
#ifndef COMMON_H
#define COMMON_H
```

```
#define ATTR_MOONSHOT_CERTIFICATE 245
#define ATTR_MOONSHOT_REQUEST 246
#define ATTR_MOONSHOT_IDPREPLY 247
#define ATTR_MOONSHOT_PROXYREPLY      248

typedef struct rlm_moonshot_t {
        char            *pub_key;
    char        *priv_key;
    char        *priv_key_password;
} rlm_moonshot_t;


#endif
```

include/idpmodule.h

```
#ifndef IDPMODULE_H
#define IDPMODULE_H

extern void idp_handle_requests(REQUEST *request);

#endif
```

include/mod_smime.h

```
#ifndef MOD_SMIME_H
#define MOD_SMIME_H

#include <openssl/x509.h>

extern int pack_mime_text(char *input, int len, char **output);
extern int unpack_mime_text(char *input, int len, char **output);

extern int pack_mime_cert(X509 *input, char **output);
extern void unpack_mime_cert(char *input, int len, X509 **output);

extern char *pack_smime_text(char *input, EVP_PKEY *pkey, X509 *pubcert);
extern char *unpack_smime_text(char *input, EVP_PKEY *pkey, X509 *cert);

#endif
```

include/proxymodule.h

```
#ifndef PROXYMODULE_H
#define PROXYMODULE_H

extern char *obtain_attributes(char *input);

#endif
```

include/request_handler_preproxy.h

```
#ifndef REQUEST_HANDLER_PREPROXY_H
#define REQUEST_HANDLER_PREPROXY_H

extern void proxy_handle_request(REQUEST *request);

#endif
```

include/x509_mod.h

```
#ifndef X509_MOD_H
#define X509_MOD_H
```

```
extern int read_public_certificate(void *instance);
extern int read_private_certificate(void *instance);

#endif
```

include/mod_base64.h

```
#ifndef MOD_BASE64_H
#define MOD_BASE64_H

extern char *base64(char *input, int length);
extern char *unbase64(char *input, int length);

//extern int base64_encode(char *input, int input_len, char **output);
//extern int base64_decode(char *input, int input_len, char **output);

#endif
```

Makefile

```
######################################################################
#
# TARGET should be set by autoconf only.  Don't touch it.
#
# The SRCS definition should list ALL source files.
#
# The HEADERS definition should list ALL header files
#
# RLM_CFLAGS defines addition C compiler flags.  You usually don't
# want to modify this, though.  Get it from autoconf.
#
# The RLM_LIBS definition should list ALL required libraries.
# These libraries really should be pulled from the 'config.mak'
# definitions, if at all possible.  These definitions are also
# echoed into another file in ../lib, where they're picked up by
# ../main/Makefile for building the version of the server with
# statically linked modules.  Get it from autoconf.
#
# RLM_INSTALL is the names of additional rules you need to install
# some particular portion of the module.  Usually, leave it blank.
#
######################################################################
TARGET      = rlm_moonshot
SRCS        = rlm_moonshot.c idpmodule.c mod_smime.c proxymodule.c
request_handler_preproxy.c x509_mod.c
RLM_CFLAGS  =  -I/usr/include -Iinclude
RLM_LIBS    =  -lc
RLM_INSTALL = install-moonshot

## this uses the RLM_CFLAGS and RLM_LIBS and SRCS defs to make TARGET.
include ../rules.mak

$(LT_OBJS): $(HEADERS)

## the rule that RLM_INSTALL tells the parent rules.mak to use.
install-moonshot:
        touch .
```