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

29 May 2013,
Version 0.1

Management summary

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 'Need to know'. 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 an SSH-shell using the user's existing credentials. This should be done without making the password or password hash known anywhere but the user's instance's RADIUS-server. This includes availability by sniffing. During the development, the focus was to follow the security principles and the creation of a standardized solution.

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 server using Radius. The priority of the project, considering the time restrictions, is the delivery of solutions that can be presented and used on an international level.

Need to know

This chapter provides an overview of the need to know subjects. The subjects are necessary in order to understand the more technical aspects of this project and the problems we have faced with the possible solutions.

EAP-TTLS

"*The Tunneled TLS EAP method (EAP-TTLS) is very similar to EAP-PEAP in the way that it works and the features that it provides. The difference is that instead of encapsulating EAP messages within TLS, the TLS payload of EAP-TTLS messages consists of a sequence of attributes. By including a RADIUS EAP-Message attribute in the payload, EAP-TTLS can be made to provide the same functionality as EAP-PEAP. If, however, a RADIUS Password or CHAP-Password attribute is encapsulated, EAP-TTLS can protect the legacy authentication mechanisms of RADIUS. The advantage of this becomes apparent if the EAP-TTLS server is used as a proxy to mediate between an access point and a legacy home RADIUS server. When the EAP-TTLS server forwards RADIUS messages to the home RADIUS server, it encapsulates the attributes protected by EAP-TTLS and inserts them directly into the forwarded message. The EAP-TTLS messages are not forwarded to the home RADIUS server. Thus the legacy authentication mechanisms supported by existing RADIUS severs in the infrastructure can be protected for transmission over wireless LANs.*"[(1)]

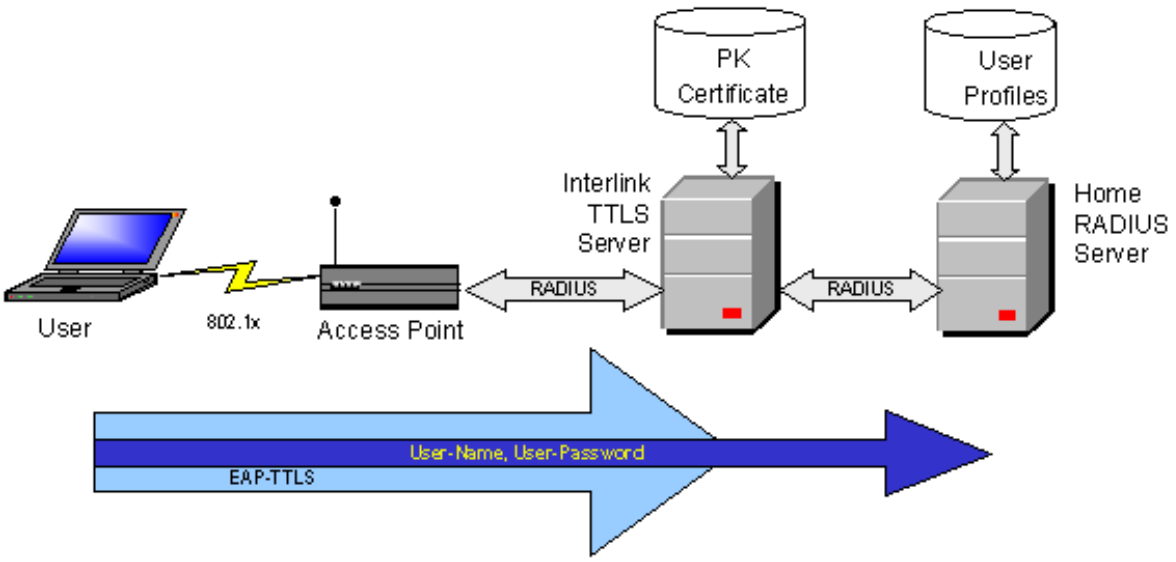


Figure 1 — How a TTLS server interacts with a legacy RADIUS server[(1)]

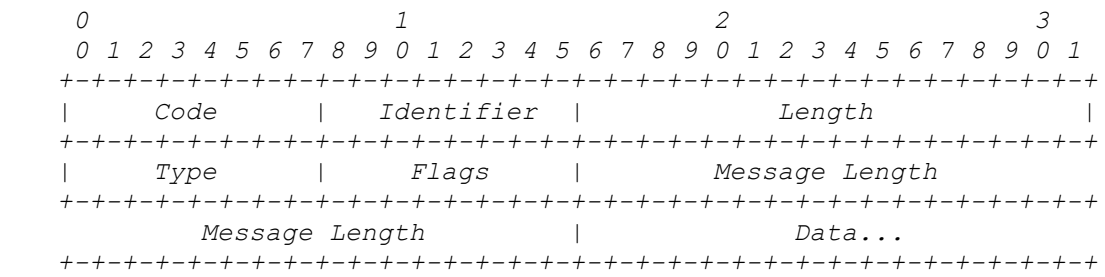
Actors
Client - A client trying to authenticate / get access to network resources

Network Access Server - Access point to the network (a switch, wireless access point, or something completely different like an SSH server).

TTLS AAA server - The server which securely (over EAP/TTLS) negotiates the authentication process for the client

AAA/H Server - The AAA (home) server with access to the clients credentials. This can be the same as the TTLS AAA server.

"***Packet encapsulation***
EAP/TTLSv1 packet:



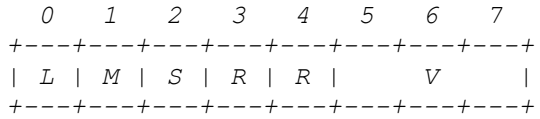
Code:
1 for request, 2 for response.

Identifier:
The Identifier field is one octet and aids in matching responses with requests. The Identifier field MUST be changed for each request packet and MUST be echoed in each response packet.

Length:
The Length field is two octets and indicates the number of octets in the entire EAP packet, from the Code field through the Data field.

Type:
21 (EAP-TTLS, all versions)

Flags:



L = Length included
M = More fragments
S = Start
R = Reserved
V = Version (001 for EAP-TTLSv1)

The L bit is set to indicate the presence of the four octet TLS Message Length field. The M bit indicates that more fragments are to come. The S bit indicates a Start message. The V bit is set to the version of EAP-TTLS, and is set to 001 for EAP-TTLSv1.

Message Length:
The Message Length field is four octets, and is present only if the L bit is set. This field provides the total length of the raw data message sequence prior to fragmentation.

Data:
For all packets other than a Start packet, the Data field consists of the raw TLS message sequence or fragment thereof. For a Start packet, the Data field may optionally contain an AVP sequence."[2]

"***Security relationships***
Client <-> NAS - No pre-existing security relationship.

NAS <-> TTLS AAA Server <-> AAA/H Server - Pre-existing security relationship is assumed. The secure connection between these points is not in the scope of the EAP/TTLS protocol and has to be configured separately. RADIUS uses pre-shared keys by default to connect with NAS devices (and other RADIUS servers, seen as NAS devices by RADIUS servers).

Client <-> TTLS AAA Server - One-way trust based on the servers CA certificate, or two-way trust if client certificate validation is enabled."[3]

Use-case: Client tries to authenticate with a TTLS RADIUS server

1. Client tries to connect to the NAS
2. NAS sends an EAP-Request/Identity message to the client
3. Client responds with an EAP-Response/Identity message (no username / password is present in this message)
4. The NAS now acts as a passthrough device, allowing the TTLS server to negotiate the EAP-TTLS with the client directly
   TLS Handshake:
5. Server sends an EAP/TTLS start packet to the client
6. Client sends **ClientHello** message with highest supported TLS version and a list of suggested ciphersuites
7. Server sends **ServerHello** message with chosen TLS version and chosen ciphersuite (EAP/TTLSv1 here)
8. Server sends certificate to authenticate itself
9. Client verifies this and generates a pre-master secret for the session, ecrypts it with the servers public key (from the server certificate) and (optionally) its own certificate to the server
10. The server (optionally) verifies the client certificate.
11. Using its private key, the server decrypts the client's pre-master secret.
12. Both the client and the server use the pre-master secret, together with a random number from both Hello messages, to compute the master secret that will be used as the symmetric key to encrypt and decrypt all future communication between server and client.
13. Both server and client send each other a message confirming that future messages will be encrypted. They then send a separate message encrypted with the new master secret, that the client / server part of the TLS handshake is finished
    AVP (attribute-value pairs) exchange
14. Using the master secret, the client now tunnels attribute-value pairs to the EAP/TTLS server, which uses this information to attempt an AAA authentication.
15. … (W.I.P.)

GSS(API)

"*The GSSAPI is an IETF standard that provides a set of cryptographic services to an application. The services are provided via a well-defined application programming interface. The cryptographic services are:*

- *Context/session setup and shutdown*
- *Encrypting and decrypting messages*
- *Message signing and verification*

The API is designed to work with a number of cryptographic technologies, but each technology separately defines the content of packets. Two independently written applications that use the GSSAPI may not be able to interoperate if they are not using the same underlying cryptographic technology.

There are at least two standard protocol-level implementations of the GSSAPI, one using Kerberos and the other using RSA public keys. In order to understand what is needed to support a particular implementation of the GSSAPI, you also need to know which underlying cryptographic technology has been used.

The GSSAPI works best in applications where the connections between computers match the transactions being performed. If multiple connections are needed to finish a transaction, each one will require a new GSSAPI session, because the GSSAPI does not include any support for identifying the cryptographic context of a message. Applications that need this functionality should probably be using TLS or SSL.

Because of the lack of context, the GSSAPI does not work well with connectionless protocols like UDP; it is really suited only for use with connection-oriented protocols like TCP."[4]

*"**Authentication Framework***

This example illustrates use of the GSS-API in conjunction with public-key mechanisms, consistent with the X.509 Directory Authentication Framework. The GSS_Acquire_cred() call establishes a credentials structure, making the client's private key accessible for use on behalf of the client. The client calls GSS_Init_sec_context(), which interrogates the Directory to acquire (and validate) a chain of public-key certificates, thereby collecting the public key of the service. The certificate validation operation determines that suitable integrity checks were applied by trusted authorities and that those certificates have not expired. GSS_Init_sec_context() generates a secret key for use in per-message protection operations on the context, and enciphers that secret key under the service's public key.

The enciphered secret key, along with an authenticator quantity signed with the client's private key, is included in the output_token from GSS_Init_sec_context(). The output_token also carries a certification path, consisting of a certificate chain leading from the service to the client; a variant approach would defer this path resolution to be performed by the service instead of being asserted by the client. The client application sends the output_token to the service. The service passes the received token as the input_token argument to GSS_Accept_sec_context(). GSS_Accept_sec_context() validates the certification path, and as a result determines a certified binding between the client's distinguished name and the client's public key. Given that public key, GSS_Accept_sec_context() can process the input_token's authenticator quantity and verify that the client's private key was used to sign the input_token. At this point, the client is authenticated to the service. The service uses its private key to decipher the enciphered secret key provided to it for per-message protection operations on the context. The client calls GSS_GetMIC() or GSS_Wrap() on a data message, which causes per-message authentication, integrity, and (optional) confidentiality facilities to be applied to that message. The service uses the context's shared secret key to perform corresponding GSS_VerifyMIC() and GSS_Unwrap() calls."[5]

*"**User Authentication with GSSAPI***
GSSAPI (Generic Security Service Application Programming Interface) is a function interface that provides security services for applications in a mechanism independent way. This allows different security mechanisms to be used via one standardized API. GSSAPI is often linked with Kerberos, which is the most common mechanism of GSSAPI." [6]

SSH

"Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network. It consists of three major components:

- *The Transport Layer Protocol [SSH-TRANS] provides server authentication, confidentiality, and integrity. It may optionally also provide compression. The transport layer will typically be run over a TCP/IP connection, but might also be used on top of any other reliable data stream.*

- *The User Authentication Protocol [SSH-USERAUTH] authenticates the client-side user to the server. It runs over the transport layer protocol.*

- *The Connection Protocol [SSH-CONNECT] multiplexes the encrypted tunnel into several logical channels. It runs over the user authentication protocol.*

The client sends a service request once a secure transport layer connection has been established. A second service request is send after user authentication is complete. This allows new protocols to be defined and coexist with the protocols listed above. The connection protocol provides channels that can be used for a wide range of purposes. Standard methods are provided for setting up secure interactive shell sessions and for forwarding ("tunneling") arbitrary TCP/IP ports and X11 connections."[7]

OpenSSH

"OpenSSH is a FREE version of the SSH connectivity tools that technical users of the Internet rely on. Users of telnet, rlogin, and ftp may not realize that their password is transmitted across the Internet unencrypted, but it is. OpenSSH encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions.

The OpenSSH suite replaces rlogin and telnet with the ssh program, rcp with scp, and ftp with sftp. Also included is sshd (the server side of the package), and the other utilities like ssh-add, ssh-agent,ssh-keysign, ssh-keyscan, ssh-keygen and sftp-server.

OpenSSH is developed by the OpenBSD Project. The software is developed in countries that permit cryptography export and is freely useable and re-useable by everyone under a BSD license. However, development has costs, so if you find OpenSSH useful (particularly if you use it in a commercial system that is distributed) please consider donating to help fund the project.

OpenSSH is developed by two teams. One team does strictly OpenBSD-based development, aiming to produce code that is as clean, simple, and secure as possible. We believe that simplicity without the portability "goop" allows for better code quality control and easier review. The other team then takes the clean version and makes it portable (adding the "goop") to make it run on many operating systems -- the so-called ***-p*** *releases, ie "OpenSSH 4.0p1".*

We sell OpenSSH T-shirts and posters. Sales of these items also help to fund development. Donations and other contributions have come entirely from end-users.

Please take note of our Who uses it page, which list just some of the vendors who incorporate OpenSSH into their own products -- as a critically important security / access feature -- instead of writing their own SSH implementation or purchasing one from another vendor. This list specifically includes companies like Cisco, Juniper, Apple, Red Hat, and Novell; but probably includes almost all router, switch or unix-like operating system vendors. In the 10 years since the inception of the OpenSSH project, these companies have contributed not even a dime of thanks in support of the OpenSSH project (despite numerous requests)."[8]

RADIUS

"RADIUS is a widely used protocol in network environments. It is commonly used for embedded network devices such as routers, modem servers, switches, etc. It is used for several reasons:

- *The embedded systems generally cannot deal with a large number of users with distinct authentication information. This requires more storage than many embedded systems possess.*
- *RADIUS facilitates centralized user administration, which is important for several of these applications. Many ISPs have tens of thousands, hundreds of thousands, or even millions of users. Users are added and deleted continuously throughout the day, and user authentication information changes constantly. Centralized administration of users in this setting is an operational requirement.*
- *RADIUS consistently provides some level of protection against a sniffing, active attacker. Other remote authentication protocols provide either intermittent protection, inadequate protection or non-existent protection. RADIUS's primary competition for remote authentication is TACACS+ and LDAP. LDAP natively provides no protection against sniffing or active attackers. TACACS+ is subtly flawed, as discussed by Solar Designer in his advisory.*
- *RADIUS support is nearly omni-present. Other remote authentication protocols do not have consistent support from hardware vendors, whereas RADIUS is uniformly supported. Because the platforms on which RADIUS is implemented on are often embedded systems, there are limited opportunities to support additional protocols. Any changes to the RADIUS protocol would have to be at least minimally compatible with pre-existing (unmodified) RADIUS clients and servers.*

RADIUS is currently the de-facto standard for remote authentication. It is very prevalent in both new and legacy systems.

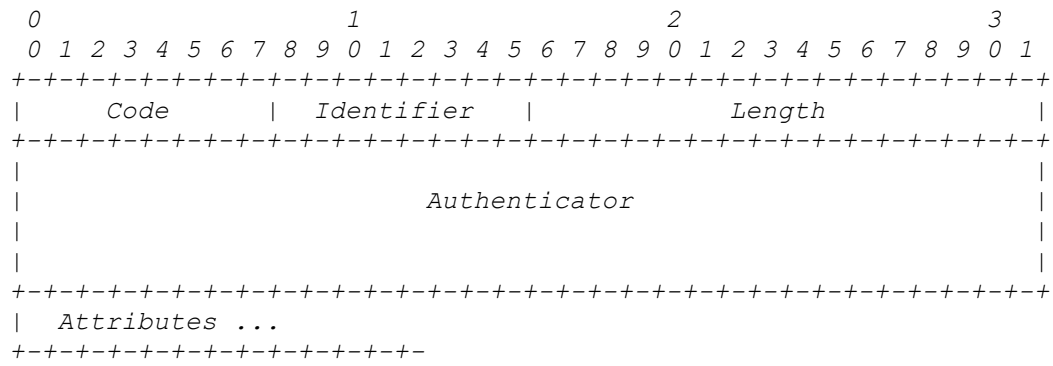
Applicability

This analysis deals with some of the characteristics of the base RADIUS protocol and of the User-Password attribute. Depending on the mode of authentication used, the described User-Password weaknesses may or may not compromise the security of the underlying authentication scheme. A complete compromise of the User-Password attribute would result in the complete compromise of the normal Username/Password or PAP authentication schemes, because both of these systems include otherwise unprotected authentication information in the User-Password attribute. On the other hand, when a Challenge/Response system is in use, a complete compromise of the User-Password attribute would only expose the underlying Challenge/Response information to additional attack, which may or may not lead to a complete compromise of the authentication system, depending on the strength of the underlying authentication system.

This analysis does not cover the RADIUS protocol's accounting functionality (which is, incidentally, also flawed, but normally does not transport information that must be kept confidential).

Protocol Summary

A summary of the RADIUS packet is below (from the RFC):



The code establishes the type of RADIUS packet. The codes are:

Value	*Description*
1	*Access-Request*
2	*Access-Accept*
3	*Access-Reject*
4	*Accounting-Request*
5	*Accounting-Response*
11	*Access-Challenge*
12	*Status-Server (experimental)*
13	*Status-Client (experimental)*
255	*Reserved*

The identifier is a one octet value that allows the RADIUS client to match a RADIUS response with the correct outstanding request.

The attributes section is where an arbitrary number of attribute fields are stored. The only pertinent attributes for this discussion are the User-Name and User-Password attributes.

This description will concentrate on the most common type of RADIUS exchange: An Access-Request involving a username and user password, followed by either an Access-Accept, Access-Reject or a failure. I will refer to the two participants in this protocol as the client and the server. The client is the entity that has authentication information that it wishes to validate. The server is the entity that has access to a database of authentication information that it can use to validate the client's authentication request.

Initial Client Processing

The client creates an Access-Request RADIUS packet, including at least the User-Name and User-Password attributes.

The Access-Request packet's identifier field is generated by the client. The generation process for the identifier field is not specified by the RADIUS protocol specification, but it is usually implemented as a simple counter that is incremented for each request.

The Access-Request packet contains a 16 octet Request Authenticator in the authenticator field. This Request authenticator is a randomly chosen 16 octet string.

This packet is completely unprotected, except for the User-Password attribute, which is protected as follows:

The client and server share a secret. That shared secret followed by the Request Authenticator is put through an MD5 hash to create a 16 octet value which is XORed with the password entered by the user. If the user password is greater than 16 octets, additional MD5 calculations are performed, using the previous ciphertext instead of the Request Authenticator.

More formally:
Call the shared secret S and the pseudo-random 128-bit Request Authenticator RA. The password is broken into 16-octet blocks p1, p2, ... pn, with the last block padded at the end with '0's to a 16-octet boundary. The ciphertext blocks are c1, c2... cn.

c1 = p1 XOR MD5(S + RA)
c2 = p2 XOR MD5(S + c1)
.
.
.
cn = pn XOR MD5(S + cn-1)

The User-Password attribute contains ***c1+c2+...+cn****, Where + denotes concatenation.*

Server Processing

The server receives the RADIUS Access-Request packet and verifies that the server possesses a shared secret for the client. If the server does not possess a shared secret for the client, the request is silently dropped.

Because the server also possesses the shared secret, it can go through a slightly modified version of the client's protection process on the User-Password attribute and obtain the unprotected password. It then uses its authentication database to validate the username and password. If the password is valid, the server creates an Access-Accept packet to send back to the client. If the password is invalid, the server creates an Access-Reject packet to send back to the client.

Both the Access-Accept packet and the Access-Reject packet use the same identifier value from the client's Access-Request packet, and put a Response Authenticator in the Authenticator field. The Response Authenticator is the is the MD5 hash of the response packet with the associated request packet's Request Authenticator in the Authenticator field, concatenated with the shared secret.

That is, ResponseAuth = MD5(Code+ID+Length+RequestAuth+Attributes+Secret) where + denotes concatenation.

Client Post Processing

When the client receives a response packet, it attempts to match it with an outstanding request using the identifier field. If the client does not have an outstanding request using the same identifier, the response is silently discarded. The client then verifies the Response Authenticator by performing the same Response Authenticator calculation the server performed, and then comparing the result with the Authenticator field. If the Response Authenticator does not match, the packet is silently discarded.

If the client received a verified Access-Accept packet, the username and password are considered to be correct, and the user is authenticated. If the client received a verified Access-Reject message, the username and password are considered to be incorrect, and the user is not authenticated.”[9]

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. We will then explain the supported algorithms. Using those pieces of information as context, we shall then describe how our module exchanges data between different Radius servers and the client.

Message formats

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

Supported Algorithms

Encryption

The Identity Provider will send RSA-encrypted messages to the Radius Proxy. The Radius Proxy will respond by signing the message with RSA and send this message to the client.

Hashing

The Radius proxy will hash the message received from the Identity Provider with SHA1 and send the message encrypted to the client.

Encoding

The bodies of the MIME-messages are encoded in Base64. The headers of the messagesare constructed with US ASCII.

Certificate format

For the certificate format, the PKCS#7 format will be used, according to the S/MIME standard.

Data exchange

The sequence diagram shows the flow of data when a request is being handled by our module.

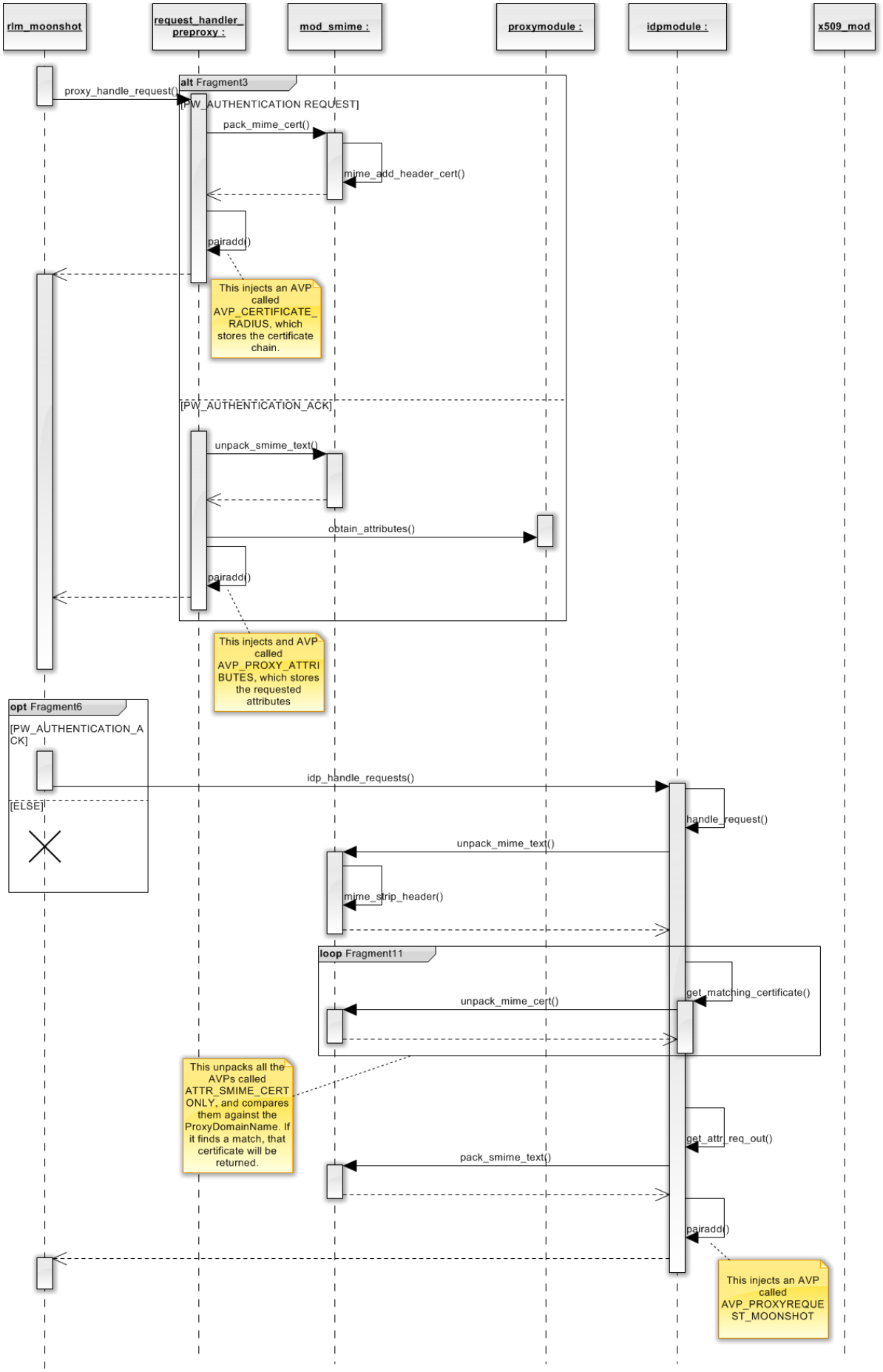
When a request arrives at the rlm_moonshot, a different function will be used depending on its position in the chain. This could be either a proxy, or identity provider. If the module is acting as a proxy, it will put the request through to the request_handler_preproxy. This will in turn det ermine if we are working with a request for authentication, or an authentication that is already a cknowledged. This is a determined from a variable in the request. The request contains a so-call ed “packet” struct, and this in turn contains a variable integer called “code”. This code is used t o determine the type of request by Radius. In our case, we are looking for either PW_AUTHEN TICATION_REQUEST or PW_AUTHENTICATION_ACK.

An authentication request will pack the server’s public certificate in a mime-message. The certi ficate-chain is then injected into the request as an attribute/value pair called AVP_CERTIFICA TE_RADIUS.

An already acknowledged authentication will extract the attribute/value pairs from the request. This is also a variable located inside the earlier mentioned packet, called “vps”. Of every attrib ute/value pair, it will then check the attributename. If this equals “ATTR_SMIME_REQUEST” , it will use the mod_smime to unpack this data, and use the proxymodule to obtain the attribute s from the resulting char pointer. These attributes are then injected into the request as an attribu te/value pair called AVP_PROXY_ATTRIBUTES.

In the case that our module is serving as an identity provider instead of a proxy, a different path is taken. It will first determine whether or not the request was accepted access, by checking the request’s “code” variable mentioned above, except this one is located in “reply” rather than “p acket”. If this equals PW_AUTHENTICATION_ACK, it will tell the idpmodule to handle the r equest. Otherwise, the request will simply be ignored.

The idp_module will first find all the attribute/value pairs in our request that have ATTR_SMI ME_REQUEST as their attributename. The value of this pair is our URN with the attribute req uests, packed in a mime-message.Of each of these attribute/value pairs it will then find out the l ength of our value pair by unpacking it from the mime-message it came in. After this is accomp lished, we should have access to a URN which we can parse into our desired structures. The res ult is an attr_req_in struct containing all the data our URN used to hold. We use this to unpack t he MIME certificates, and find the attributes. We create a URN that holds all these attributes an d pack it in an S/MIME message using a public key we received from one of the proxies, and in ject this into our request as an attribute/value pair called AVP+PROXYREQUEST_MOONSH OT.



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 to clarify different parts in our story.

Dependencies

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

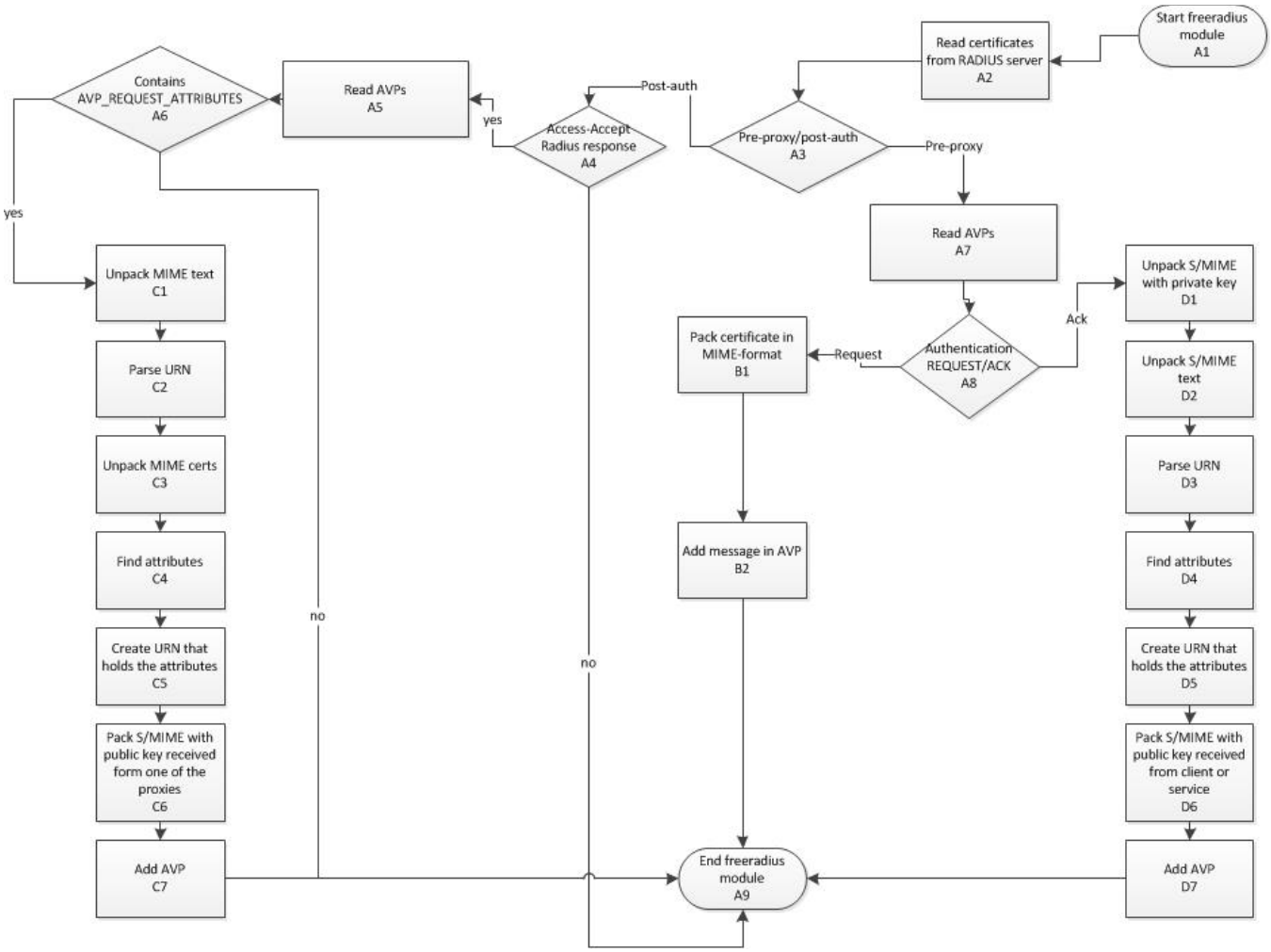
- FreeRADIUS 2.1.12
- OpenSSL 1.0.0-fips
- make
- autoconf
- gcc
- wget

We haven't tested other versions of the above programs and therefore the current version of this module will not this.

Story

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 diagram underneath.



File diagram

The FreeRadius module, counting only our created content, counts six source files, eigth header s, and four structs. The source files are rlm_moonshot.c, mod_smime.c, request_handler_prepro xy.c, proxymodule.c, idpmodule.c and x509_mod.c. There is one header per source file, in add ition to a config.h and common.h header. The structs are attr_req_in, attr_req_out, avp_struct, a nd 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 iden tity 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.

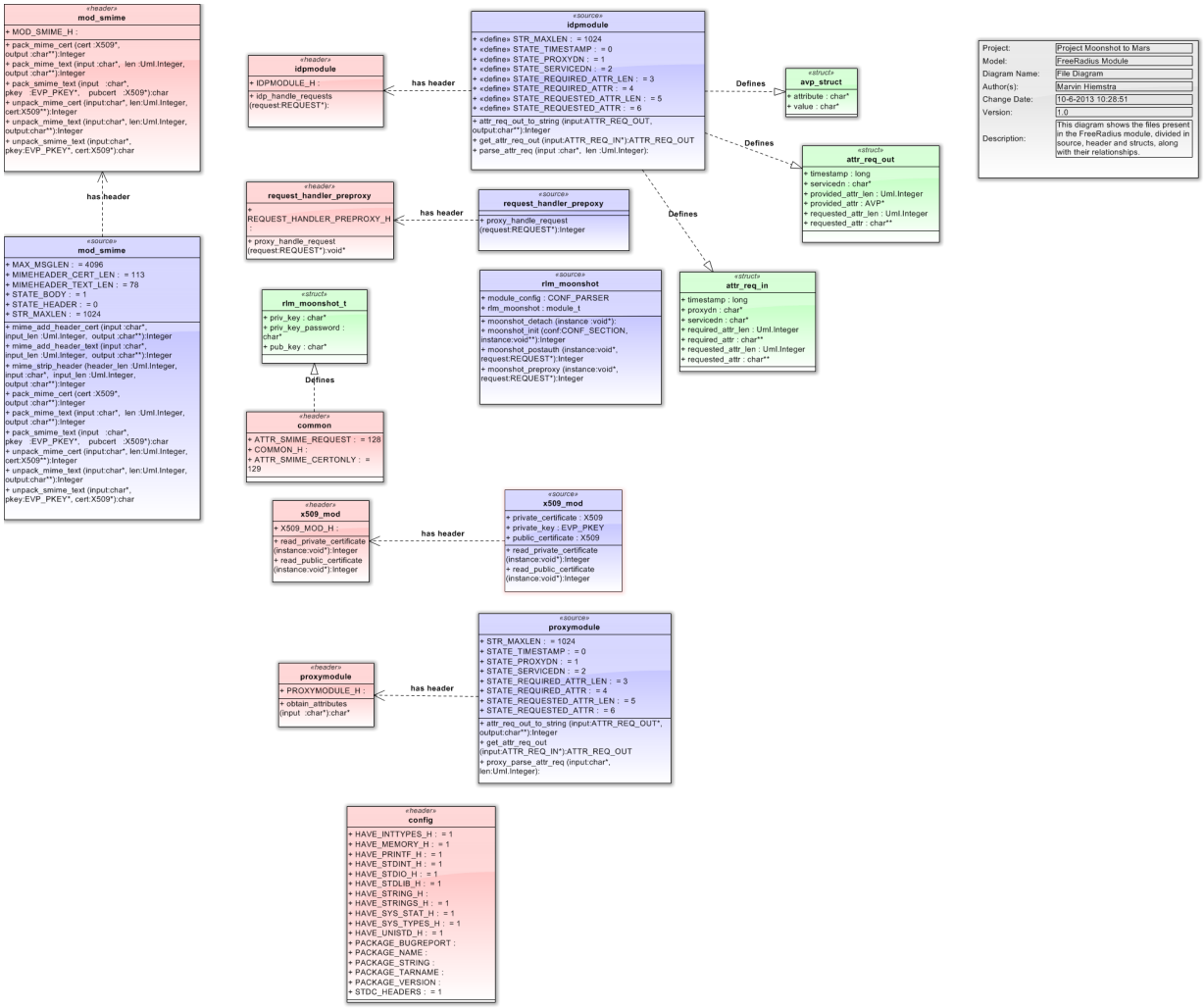
request_handler_preproxy.c is the start of the path from client to proxy and proxy to identity pr ovider. It determines if we are dealing with a request for authorization, or an already acknowle dged authentication.

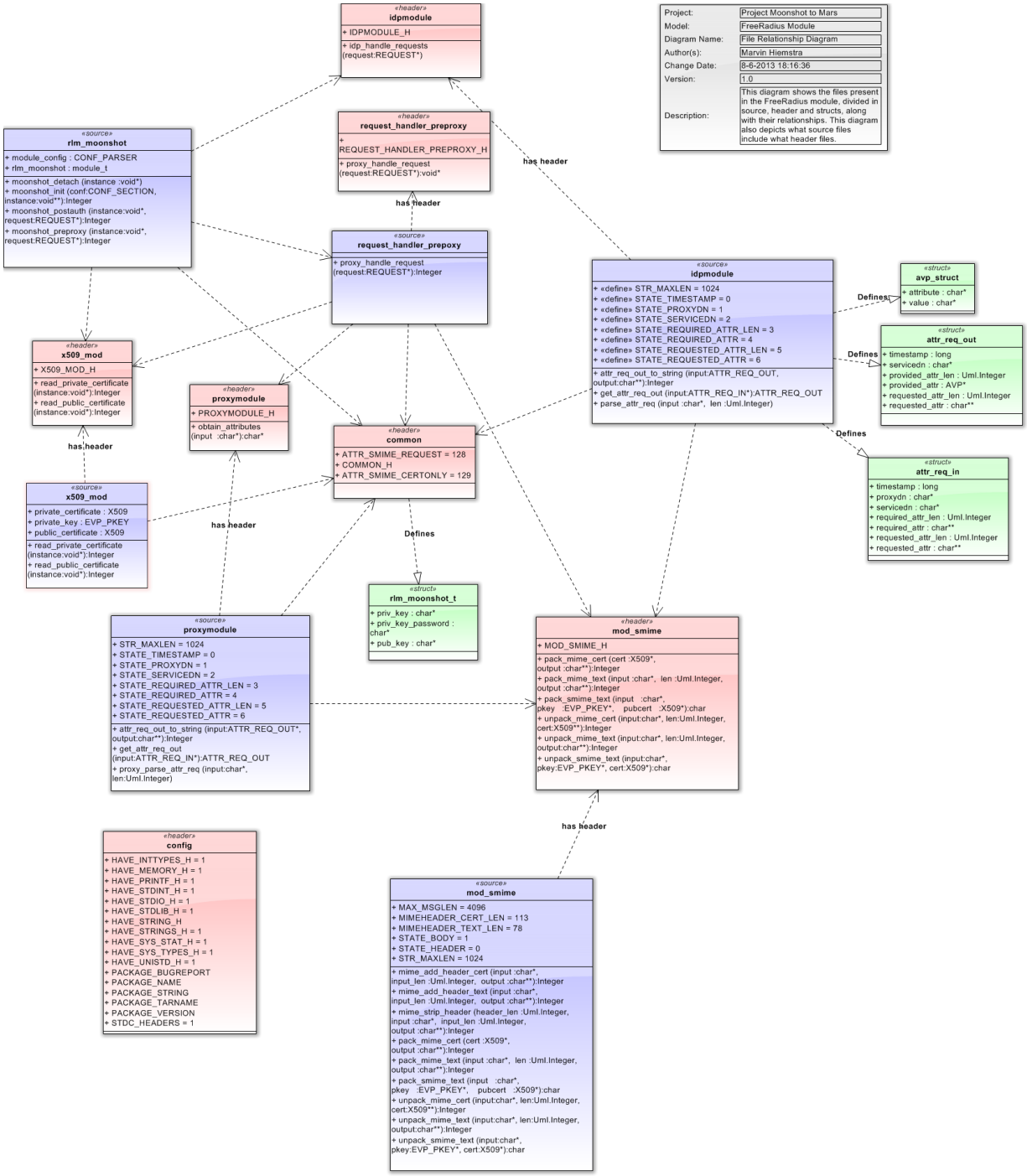
proxymodule.c and idpmodule.c are each other's counterparts, with roughly the same goals, ada pted to their respective environment. The biggest difference is that in proxymodule.c, the reque sted attribute/value pairs are still unknown. Therefore it will extract these from the URN, reque st the obtained 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 publi c and private certificates.We also have a header called common.h that is included in all source f iles(except for mod_smime.c). This header is responsible for defining the rlm_moonshot_t stru ct. 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 t he encryption and decryption of the attribute/value pairs.

avp_struct contains two char pointers, representing an attributename and a value. attr_req_in an d 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 ser vice.

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

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





Known issues

General:

- The way AVPs are split into and joined from 255-byte pieces is makeshift and currently doesn't support handling more than one attribute request per RADIUS packet
- The entire module is going through the last round of bugfixes. A partially working Proof of Concept is present at the moment and a fully working demonstration is expected to be possible within one week

idpmodule.c:

- Parsing or copying the ServiceDN URN element is done incorrectly. More problems in the parser or struct constructor functions are expected
- get_matching_certificate does not yet attempt to reconstruct the certificate mime message out of multiple AVPs, but rather expects the certificate to be present in a single AVP.

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

Source code

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

rlm_moonshot.c

```
#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, off-
setof(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 mod-
ule_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;
}
```

```
static int moonshot_preproxy(void *instance, REQUEST *request)
{
        /* quiet the compiler */
        instance = instance;

    //Handle pre-proxy requests, this is done by request_handler_preproxy.c
    proxy_handle_request(request);

        return RLM_MODULE_OK;
}

static int moonshot_postauth(void *instance, REQUEST *request)
{
        /* quiet the compiler */
        instance = instance;

        //Is it an Access-Accept?
        if (request->reply->code == PW_AUTHENTICATION_ACK)
        {
                idp_handle_requests(request);
        }

        return RLM_MODULE_OK;
}

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 */
                NULL,                   /* post-proxy */
                moonshot_postauth       /* post-auth */
        },
};
```

idpmodule.c

```
#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;

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

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;

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;

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;
}

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;
}

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;
}

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);
}

static X509 *get_matching_certificate(REQUEST *request, char *dn)
{
        X509 *tmp_cert;

        VALUE_PAIR *vp = request->packet->vps;
        do
        {
                if (vp->attribute == ATTR_MOONSHOT_CERTIFICATE)
                {
                        unpack_mime_cert((char *)vp->data.octets, vp->length,
&tmp_cert);

                        if (strcmp(tmp_cert->name, dn) == 0)
                        {
                                return tmp_cert;
                        }
                        free(tmp_cert);
                }
        } while ((vp = vp->next) != 0);
        return NULL;
}

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;
        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);
        smime_msg = pack_smime_text(output_data, private_key, cert);
        for (i = 0; i <= (strlen(smime_msg) / 250); i++)
        {
                memcpy(substr, &smime_msg[i * 250], i == (strlen(smime_msg) /
250) ? strlen(smime_msg) % 250 : 250);
                avp_smime = pairmake("Moonshot-Request", substr, T_OP_EQ);
                pairadd(&request->reply->vps, avp_smime);
        }
        return;
}

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, vp->data.octets);
                }
        } while ((vp = vp->next) != 0);
        if (found)
        {
                handle_request(request, message);
        }
}
```

request_handler_preproxy.c

```
//
//  request_handler_preproxy.c
//
//
//  Created by W.A. Miltenburg on 15-05-13.
//
//

#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;

int proxy_handle_request(REQUEST *request)
{
        char message[4096];
        int found = 0;
        int i;
        char *cert_message;
        char substr[251];
        VALUE_PAIR *vp;
        switch (request->packet->code) //it's allowed to handle multiple re-
quests, 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++)
                {
                        memcpy(substr, &cert_message[i * 250], i ==
(strlen(cert_message) / 250) ? strlen(cert_message) % 250 : 250);
                        substr[250] = '\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 ba-
sically saying that our AVPs are updated

        case PW_AUTHENTICATION_ACK:
                        memset(message, 0, 4096);

            vp = request->packet->vps;

            do {
```

```
                if (vp->attribute == ATTR_MOONSHOT_REQUEST) //detect if
AVP_PROXY_REQUEST is sent by the idp module
                {
                    found = 1;
                                        strcat(message, vp->data.octets);
                }
            } while ((vp = vp -> next) != 0);
                       if (found)
                       {
                                char *message_attributes = un-
pack_smime_text((char *)vp->data.octets, private_key, private_certificate);
                                char *out_message = ob-
tain_attributes(message_attributes);
                                VALUE_PAIR *avp_attributes;

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

mod_smime.c

```
#include "crypto/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

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;
}

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) * MIME-
HEADER_TEXT_LEN) + 1);
        strcpy(*output, header);
        strcat(*output, input);
        return input_len + MIMEHEADER_TEXT_LEN + 1;
}

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;
}

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;
}

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);

        *output = unbase64(base64_out, strlen(base64_out));
        return strlen(*output);
}

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_mem_buf(outbuffer, -1);
        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_out, &bptr);
        outbuffer = strdup(bptr->data, bptr->length);

        mime_add_header_cert(outbuffer, strnlen(outbuffer, 5120), output);
        free(outbuffer);
        return 0;
}

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;
}

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;

   //Prepare general stuff
   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;
}

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)
        {
                printf("dectext: error creating bio_in, bio_dec or
bio_out\n");
                exit(1);
        }

        cms = SMIME_read_CMS(bio_in, NULL);
        if (!cms)
        {
                printf("Error parsing message to CMS\n");
                exit(1);
        }

        if (!CMS_decrypt(cms, pkey, cert, NULL, bio_out, 0))
        {
                printf("Error decrypting message\n");
                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

```
#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;

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;

typedef struct attr_req_out
{
        unsigned long timestamp;
        char *servicedn;
        int requested_attr_len;
        AVP *requested_attr;
} ATTR_REQ_OUT;

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;
}
```

```
ATTR_REQ_IN *proxy_parse_attr_req(char *input, int len)
{
        ATTR_REQ_IN *tmp_attr_req = rad_malloc(sizeof(ATTR_REQ_IN));

        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';

                                        memcpy(tmp_attr_req->provided_attr +
(attr_p * sizeof(AVP)), atoavp(item_tmp), sizeof(AVP));
                                        //tmp_attr_req->provided_attr[attr_p]
= atoavp(item_tmp);
                input_cur++;
                bzero(item_tmp, sizeof(char) * STR_MAXLEN);
                item_cur = 0;

                                        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 = real-
loc(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;
}

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;
}

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;
}

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, in-
put->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);
}

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

```
//
//  x509_mod.c
//
//
//  Created by W.A. Miltenburg on 03-06-13.
//
//

#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 pass-
word 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;
}
```

crypto/mod_base64.c

```
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <string.h>

char *base64(const unsigned char *input, int length)
{
BIO *bmem, *b64;
BUF_MEM *bptr;

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);

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

BIO_free_all(b64);

return buff;
}

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_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

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/crypto/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
```