**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

[Project Definition 5](#_Toc357002313)

[Goal 5](#_Toc357002314)

[Project conditions and limitations 5](#_Toc357002315)

[Need to know 6](#_Toc357002316)

[EAP-TTLS 6](#_Toc357002317)

[GSS(API) 10](#_Toc357002318)

[SSH 11](#_Toc357002319)

[RADIUS 13](#_Toc357002320)

[Research JaNET Moonshot 17](#_Toc357002321)

[How it works 17](#_Toc357002322)

[Applied Modifications 17](#_Toc357002323)

[Useable Parts 17](#_Toc357002324)

[Known Issues 17](#_Toc357002325)

[FreeRadius Module 18](#_Toc357002326)

[Certificate Management 18](#_Toc357002327)

[SMIME Module 18](#_Toc357002328)

[Carrier Protocol 18](#_Toc357002329)

[Supported Algorithms 18](#_Toc357002330)

[Conclusion 19](#_Toc357002331)

[Sources 20](#_Toc357002332)

[Appendix 21](#_Toc357002333)

[Output 21](#_Toc357002334)

[Overview Changed Files Openssh 22](#_Toc357002335)

**Introduction**

This Research document exists in order to create an overview of the research done by ITopia. The need for this document comes from the request from Nikhef to create an overview of the research done of JaNET’s moonshot project and the creation of ITopia’s solution to improve security of Nikhef’s own moonshot version.

In the first chapter, Project Definition, de project is explained en described. For this project study of different protocols is necessary for example SSH, GSS API, EAP-TTLS, etc. These protocols are, with the most relevant information, described is in the chapter “Need to know”. In “Research JaNET Moonshot” the results of JanNET’s moonshot with the known problems is written. Furthermore, in chapter “FreeRadius Module” our solution to make Nikhef’s moonshot project more secure is described. The last chapter will be our conclusion of our research.

# 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

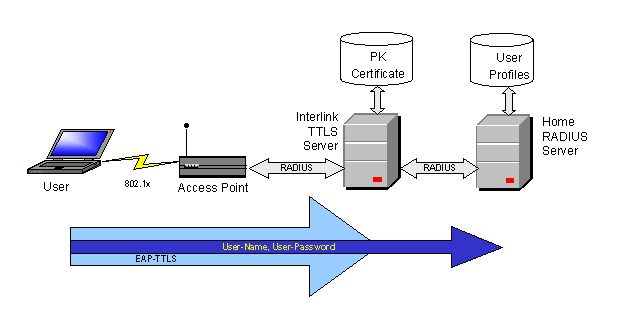
The scope of the project was to study the existing literature and presenting the findings and the development of possible solutions that will enable the user to securely connect to an OpenSSH server using Radius. The priority of the project, considering the time restrictions, is the delivery of solutions 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:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Code | Identifier | Length |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Type | Flags | Message Length

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Message Length | Data...

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

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:

0 1 2 3 4 5 6 7

+---+---+---+---+---+---+---+---+

| L | M | S | R | R | V |

+---+---+---+---+---+---+---+---+

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:**

1. Server sends an EAP/TTLS start packet to the client
2. Client sends **ClientHello** message with highest supported TLS version and a list of suggested ciphersuites
3. Server sends **ServerHello** message with chosen TLS version and chosen cipersuite (EAP/TTLSv1 here)
4. Server sends certificate to authenticate itself
5. 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
6. The server (optionally) verifies the client certificate.
7. Using its private key, the server decrypts the client’s pre-master secret.
8. 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.
9. 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**
10. 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.
11. … (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](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh&sektion=1) program, rcp with [scp](http://www.openbsd.org/cgi-bin/man.cgi?query=scp&sektion=1), and ftp with [sftp](http://www.openbsd.org/cgi-bin/man.cgi?query=sftp&sektion=1). Also included is [sshd](http://www.openbsd.org/cgi-bin/man.cgi?query=sshd&sektion=8) (the server side of the package), and the other utilities like [ssh-add](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-add&sektion=1), [ssh-agent](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-agent&sektion=1),[ssh-keysign](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-keysign&sektion=8), [ssh-keyscan](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-keyscan&sektion=1), [ssh-keygen](http://www.openbsd.org/cgi-bin/man.cgi?query=ssh-keygen&sektion=1) and [sftp-server](http://www.openbsd.org/cgi-bin/man.cgi?query=sftp-server&sektion=8).

OpenSSH is developed by [the OpenBSD Project](http://www.openbsd.org/). 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](http://www.openssh.org/donations.html).

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](http://www.openssh.org/tshirts.html) and [posters](http://www.openbsd.org/orders.html#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](http://www.openssh.org/users.html) 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):

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Code | Identifier | Length |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| |

| Authenticator |

| |

| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Attributes ...

+-+-+-+-+-+-+-+-+-+-+-+-+-

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)

# Research JaNET Moonshot

Yadayadayada

## How it works

Yada yada yada

## Applied Modifications

Yada yada yada

## Useable Parts

Yada yada yada

## Known Issues

Yada yada yada

# FreeRadius Module

Yadayadayada

## Certificate Management

yadyadayada

## SMIME Module

Yada yada yada

## Carrier Protocol

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 IdP will send messages encrypted with RSA to the Radius proxy. The Radius proxy will respond by signing the message with RSA and send the message to the client.

**Hashing**

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

**Encoding**

The mime body’s will be encoded Base64, the headers will be constructed with US ASCI.

**Certificate format**

For the certificate format, the PKCS#7 format will be used according to the SMIME standard.

# Conclusion

Yada yada yada

# Sources

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

(2) [https://tools.ietf.org/html/rfc5281](https://tools.ietf.org/html/rfc5281%20Chapter%209.1)  Chapter 9.1

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

(4) <http://docstore.mik.ua/orelly/networking_2ndEd/fire/ch14_08.htm> Whole page

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

(6) <http://www.ssh.com/manuals/server-admin/44/User_Authentication_with_GSSAPI.html> Whole page

(7) <http://www.ietf.org/rfc/rfc4251.txt> Introduction

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

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

# Appendix

## Output

## Overview Changed Files Openssh