MEAP,
MODIFIED EAP PROTOCOL FOR WLAN AUTHENTICATION

By

Xiaoying Guo

Bachelor of Computer Science
China University of Geosciences, Wuhan, China, 1997

A thesis
Presented to Ryerson University
In partial fulfillment of the
Requirements for the degree of
Master of Applied Science
In the Program of
Computer Networks

Toronto, Ontario, Canada, 2006

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

Xiaoying Guo,

MEAP, Modified EAP protocol for WLAN authentication

M.A.Sc, Computer Networks, Ryerson University, 2006

Wireless networking is becoming increasingly popular. However, the use of Wireless Local Area Networks (WLAN) also creates many security issues that do not exist in a wired world. WLAN connection no longer requires cable. Instead, data packets are sent on the air and are available to anyone with the ability to intercept and decode them. Traditional physical security measures like firewalls and security guards are less efficient in this new domain.

The IEEE has organized the 802.11i Task Group to address 802.11 security. To solve the user authentication problem, it adopted 802.1x standard. The standard relies on Extensible Authentication Protocol (EAP) to provide the authentication function. However, after the basic EAP authentication process, the Access Point (AP) still needs to be authenticated by the client. In order to implement mutual authentication, Protected EAP (PEAP), EAP-Transport Layer Security (TLS), EAP-Tunneled TLS (TTLS) or other variants of EAP are developed. But some still weak in the authentication procedure. In this thesis, a new authentication mechanism called Modified EAP (MEAP) is proposed. MEAP is based on PEAP. MEAP adds a TLS layer on top of EAP, and then uses the resulting TLS session as a tunnel to protect the later Simple Password-authenticated Exponential Key Exchange (SPEKE), which is a strong password method. MEAP can provide mutual authentication to satisfy strong authentication requirement in WLAN.

Acknowledgements

First and foremost I would like to sincerely thank my advisor Dr. Chul Kim for all the guidance and help he took in the progress of my study.

I am extremely grateful to Dr. Bobby Ma for his valuable suggestions and advise during this thesis. I am also thankful to the thesis examiners.

Many thanks to those people I love. Thanks for all their love, affection and blessings without which I would not have gotten this far in my life.

Table of Content

List of Abbreviations

AES	Advanced Encryption Standard
AP	Access Point
CA	Certification Authority
CHAP	Challenge Handshake Authentication Protocol
CCMP	Counter Mode CBC-MAC Protocol
DoS	Denial of Service
EAP	Extensible Authentication Protocol
EAPOL	EAP Over LAN
IAS	Internet Authentication Service
IV	Initialization Vector
MAC	Media Access Control
MD5	Message-Digest algorithm
MIC	Message Integrity Check
MITM	Man-In-The-Middle
MSCHAP	Microsoft Challenge Handshake Authentication Protocol
MSK	Master Session Key
NAI	Network Access Identifier
NAS	Network Access Server
NIC	Network Interface Cards
PEAP	Protected EAP
PKI	Public Key Infrastructure
PPP	Point-to-Point Protocol
RADIUS	Remote Authentication Dial-In User Service
RC4 PRNG	Ron's Code 4 Pseudo Random Number Generator
RLM	RADIUS Load Module

SPEKE	Simple Password-authenticated Exponential Key Exchange
SSL	Secure Socket Layer
TKIP	Temporal Key Integrity Protocol
TLS	Transport Layer Security
TLV	Type-Length-Value
TTLS	Tunneled Transport Layer Security
UDP	User Datagram Protocol
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WPA	Wi-Fi Protected Access

List of Tables

List of Figures

Chapter 1. Introduction

IEEE standard 802.11 is a series of specifications for Wireless Local Area Networks (WLAN). The goal of 802.11 is to define an Ethernet-like communication channel using radio waves instead of cables.

1.1 Wireless security requirements

As the growing of WLAN, it provides the users with a significant mobility advantages. However, WLAN also present some serious security challenges. In general, the challenges can be classified into three areas: authentication, encryption, and data integrity.

1.1.1 Authentication

With wireless networks, there is no physical access to the network infrastructure. There is an access point, and a client can associate with the access point without the knowledge of its location. For this reason, users need to ensure that they are connecting to legitimate access points that are part of the organization's network, not "rogue" access points. Thus, not only the user has to be authenticated by the network, but the network also needs to be authenticated by the user. This is called mutual authentication [28].

1.1.2 Encryption

Wireless networks are much easier to be tapped because physical access to the network does not require physical access to the equipment since transmissions are broadcasted over radio waves. Frames can be easily intercepted in transit by using wireless network analysis softwares. All the communication between the access points and the stations need to be encrypted.

1.1.3 Data integrity

Since a third party can intercept the data, there is a risk of malicious modification of the data. The receiver would have no way of knowing whether the data received is the original data sent or whether the data has really been sent by the specified sender. So, a mechanism to ensure integrity is necessary.

1.2 WEP-Based Security

Wireless transmissions are easier to intercept than transmissions over wired networks.
The 802.11 standard currently specifies the Wired Equivalent Privacy (WEP) security protocol to provide encrypted communication between the client and an Access Point (AP). WEP employs a symmetric key encryption algorithm based on Ron's Code 4 Pseudo Random Number Generator - RC4 PRNG (NOTE: RC4 is a stream cipher where a seed is used as input to the RC4 PRNG which produces an output bit string that is XORed with the plaintext to produce the cipher-text)[1].

Under WEP, all clients and APs on a wireless network typically use the same key to encrypt and decrypt data. The key resides in the client's computer and in each AP on the network. The 802.11 standard does not specify a key-management protocol, so all WEP keys on a network usually must be managed manually unless they are used in conjunction with a separate key-management protocol. For example, 802.1X (discussed later in this thesis) provides WEP key management. Support for WEP is standard on most current 802.11 cards and APs. WEP specifies the use of a 40-bit encryption key and there are also implementations of 104-bit keys. The encryption key is concatenated with a 24-bit Initialization Vector (IV), resulting in a 64- or 128-bit key. This key is input into a pseudorandom number generator. The resulting sequence is used to encrypt the data to be transmitted. WEP keys can be entered in alphanumeric text or hexadecimal form.

1.2.1 WEP's Weakness

WEP's initial goal was to provide a level of security that conformed to the difficulty of tapping Ethernet network traffic. In the case of wired Ethernet, you would need a physical access to a network to sniff packets and intercept data. WEP's minimal security should have met at least that level of protection. Unfortunately, WEP could not meet such security requirements because of flaws in the conception and implementation of the protocol.

Several articles appeared in 2001[25] [26] to address the weaknesses of the WEP. Initially, cracking WEP required some expertise. But widely available, simple-to-use software now makes it a snap for even a casual cracker to extract a WEP key from a home or business network. Home users with less-busy Wi-Fi networks are less likely to be cracked both for reasons of intent--someone might not bother--and time. The fewer the packets, the longer it takes to crack a network and gain access to the traffic passing over it.

The vulnerability of WEP can be summarized as the following [1][20]:

- WEP key recovery: WEP uses the same WEP key and a different IV to encrypt data. An IV is used to alter the key stream. The IV is a numeric value that is concatenated to the base key before the key stream is generated. Every time the IV changes, so does the key stream. Because the IV has only a limited range (0, 2^{24}), the same IVs may be used over and over again. By picking the repeating IVs out, an attacker can easily crack the WEP key.
- Unauthorized decryption and the violation of data integrity: Once the WEP key is revealed, the attacker can transform the cipher text into its original form. Based

on the understanding of the algorithm, a hacker may use the cracked WEP key to modify the cipher text and forward the changed message to the receiver.

- One-way authentication: WEP only provides a method for the clients to authenticate AP, not a mutual authentication. As a result, it is possible for an attacker to reroute the data to the AP through an unauthorized path.

Corporations relying on 802.1X network authentication systems that can automatically swap WEP keys after just a certain number of packets may still have security problems. Because more recently discovered methods of cracking WEP keys reduce the threshold for data interception down below the number of packets sent before the authentication system changes the key.

1.2.2 WEP's Replacement

The IEEE task group that was responsible for security, 802.11i, developed a compromise solution that looked backwards to fix WEP and forwards to replace it without losing compatibility [13]. The backward compatible solution is called Temporal Key Integrity Protocol (TKIP).

TKIP has the following features:

- Create a longer, better IV of 48 bits than WEP;
- Increase randomness;
- Use a master key from which other keys are derived;
- Mix keys and IVs in such a way that each packet has its own unique key.

The 802.11i [2] also fixes the packet integrity by using a more advanced method of detecting tampering, and putting this information in the encrypted part of the frame instead of sending it in the clear text form.

The forward-looking part of 802.11i adds AES-CCMP (Advanced Encryption Standard - Counter Mode CBC-MAC Protocol) for an essentially impregnable hiding of data that supports longer and a cryptographically more secure stream of data than TKIP. AES is quite widely used and has been adopted by the US government. The specific AES type included in 802.11i, CCMP, is the same length as TKIP keys: 128 bits. However, its underlying algorithm is much stronger. Most Wi-Fi chips released in late 2002 and beyond include the necessary support for AES.

The TKIP and AES focus more on the encryption part of WLAN. To ensure the correct authentication, 802.1x is proposed. In the implemented form of 802.11i specification includes the support for the 802.1X and Extensible Authentication Protocol (EAP) protocols. 802.1X is a way of defining roles so that a client can connect to the access point and has a limited access -- a client can only talk to the access point, but cannot see the rest of the network until the access point queries the client and relays its messages back and forth to an authentication server, which then confirms the client's identity. EAP is the protocol used to carry the authentication information. 802.1x and EAP will be discussed in detail in Chapter 2.

1.3 The popular attacks against current WLAN

There are many related attacks such as free-loading, accidental threats, rogue WLANs and eavesdropping [16]. This thesis addressed the security related to MITM, session hijack, and spoofing.

- **Man-In-The-Middle**

  A Man-In-The-Middle (MITM) attack [17] is when an attacker tricks a client into believing that he is the entity that the client wants to connect to, in this case an access point. The attacker then takes the authentication information it receives from the client and logs on to a genuine AP and establishes himself as the man in the middle.

The attacker can then eavesdrop on the traffic or simply hijack the session. This attack has been a problem especially in tunneled EAP methods such as EAP-Transport Layer Security (TLS), EAP-Tunneled TLS (TTLS) and EAP-Protected EAP (PEAP) [4]. In order for this attack to be successful, the authentication process needs to be one-sided, i.e. only the client is authenticated and not the AP. In order for eavesdropping to work, the authentication needs to be simple enough so that the attacker can decode and install the session key. Only mutual and robust authentication can prevent these types of attacks.

- **Session Hijack**

This type of attack is used by the attacker to gain control of a legitimate user's session and use it as its own [10]. It is often used in conjunction with MITM and an attack scenario using a challenge-response protocol can be illustrated as follows: An attacker connects to an AP and receives a challenge. The attacker then poses as an AP to get a client to connect to it. When a client connects to the attacker, it gets the same challenge as that of the attacker. The client encrypts it, and then sends it back to the attacker. The attacker then forwards the response to the genuine AP and is allowed to log on. As a final act, the attacker sends a logoff-message to the client that has no reason to suspect foul play. The necessary conditions that could be used to prevent this attack are the same as MITM.

- **Spoofing**

Spoofing is when an attacker fakes the origin of a packet in order to achieve something. It can also assume a different identity by switching to a different medium access control (MAC) address. If an AP only uses MAC association as authentication method this is a very trivial way to gain access to the network. MAC association means that the AP has a list over the MAC addresses that are allowed; packets sent from others are just thrown away. This technique is often used to facilitate other attacks, for instance MITM, where the attacker uses spoofing to impersonate an AP to lure clients to connect to it.

The spoofing itself is hard to stop, as there exists no network-level method of ascertaining that an entity actually is who he says he is. But spoofing on its own is rarely a major problem: it has to be combined with some other kind of attack in order to be really damaging.

- **Denial of Service**

  Denial of Service (DoS) is exactly what it sounds like: it is about denying someone access to a service. It can be as easy as setting up a jamming beacon to disrupt the AP's signals to deny them access to the network. There are basically two kinds of DoS attacks: the ones that exploit flaws in the protocols and the ones that use brute force to overwhelm the resource. There is no universal method for this category of attacks. Tools designed to cope with them most often analyze traffic and then act on anomalous events. But it is hard since some attacks are very subtle and problematic to detect. Having a robust system that is deployed correctly using proven security protocols goes a long way against mitigating the DoS threat.

1.4 The Different Links Involved

This section indicates the different links in the authentication system. It will look at the protocols that are responsible for transport over the links and what level of security they provide. The mutual authentication will cover all of the listed links.

1.4.1 Link between the Client and the Access Point

This link is the first to be initiated before authentication takes place. It is a very tricky link to protect. Without the exchanging encryption keys in this link, there looks like can be no protection.

The protocol used on this link is EAPOL (EAP Over LAN), which has no built-in security measures except for the EAPOL-Key frame type. EAPOL-Key frames are used to transport keying information to the client after authentication. They use RC4 for encryption. They are also used with slight modifications (see the 802.11i standard for details) to protect the traffic during the four-way handshake in 802.11i. Earlier, in connection with WEP, it was mentioned that RC4 has documented the WEP's weaknesses. In this case, however, the key is of adequate length (the EAPOL-Key Encryption Key) and the amount of information is so small that the risk of exposure is negligible.

The possible attacks on this link are mainly DoS attacks. One potential attack is for an attacker to spoof the MAC address of any client trying to log on to the AP and then send a EAPOL-Logoff message. The AP will then immediately shut down the connection, shutting the client out. If this is done continuously, then the attacker will have the AP entirely to himself.

1.4.2 Authentication link between Client and Authentication Server through AP

This link is the second one (from the client's perspective) to be established and it only exists during authentication. The protocol used for this link is EAP, but the actual traffic is encapsulated within other protocols during transport. From the client to the AP, traffic is encapsulated in EAPOL packets. From the AP to the Authentication Server, traffic is embedded into the Remote Authentication Dial-In User Service (RADIUS) packets.

There are no native security mechanisms in the EAP protocol. Such mechanisms are left entirely to the implemented different authentication schemes. There is an RFC draft detailing security requirements for EAP methods [11], according to which an EAP method must support the following criteria:

- Session keys must be generated, rather than transported.
- The effective key strength must be no less than 128 bits.
- It must support mutual authentication.
- The EAP peer and server must share the same state at all times during the authentication process.
- It must be resistant to dictionary attacks.
- It must have protection against MITM attacks.
- Cipher suite negotiation must be protected.

1.4.3 Link between Access Point and Authentication Server

As RADIUS is the de facto a standard for authentication servers, the protocol for this link is RADIUS. This protocol specifies that each Access-Request packet should contain a nonce, called Request Authenticator, ensuring replay protection. This Request Authenticator is then calculated into a Message Integrity Check (MIC), called Response Authenticator, used to ensure integrity for the Access-Accept and Access-Challenge messages. To ensure confidentiality, a Keyed Hash (MD5) is used as a stream cipher to encrypt the password field. The shared RADIUS secret is used as key. There are also extensions to the protocol defined in RFC 2869 [12]. Security wise, there are a few additions, most notably a MIC for all messages. The RFC also elaborates a little more on the security issues and discusses possible attacks. For instance, if the MIC is not used the protocol is vulnerable to session hijacking, MITM and other attacks [12].

1.5 My contribution

802.11i is targeted solely for physical to link layer security. The upper layer mechanisms are needed for creating authorized and encrypted communication between the client and the actual server to be used. EAP is used as the authentication protocol. EAP separates the message exchange from the process of authentication by providing an independent exchange layer. Because of this property, EAP can be easily extended to support the other security features. For this, different variations of the EAP are proposed. But each EAP protocol has its own limitations.

EAP-TLS is a well-known protocol adopted by many WLAN users. But it needs digital certificates on both server and client side, which makes it complicated in large-scale enterprise. A more popular approach is PEAP. PEAP is a two-stage authentication method. The first stage establishes a TLS session to the server and allows the client to authenticate the server using the server's digital certificate. The second stage requires a second EAP method tunneled inside the PEAP session to authenticate the client to the RADIUS sever. This allows PEAP to use a variety of client authentication methods. MS-CHAPv2 has been the most popular approach but it has its limitation that only supports windows operation system. To support PEAP in other platform, Simple Password-authenticated Exponential Key Exchange (SPEKE) is adopted in this thesis.

The rest of the thesis is organized as follows. In chapter 2, we focus on the current authentication protocols. Here the details of the EAP authentication progress are discussed. In chapter 3, the MEAP protocol is proposed along with two parts: PEAP and SPEKE. In chapter 4, we present the MEAP implementation procedures. In chapter 5, three testing scenarios and the expected results are described. Finally, in the last chapter 6, the conclusion and further work are discussed.

Chapter 2. The current EAP authentication protocols

As we introduced in the previous chapter, WEP security features proved to be insufficient to protect wireless LAN communication. To address the security issues of the original IEEE 802.11 standard, the following additional technologies are used:

- The 802.1x standard: This protocol is used to do the authentication in WLAN. IEEE 802.1X provides per-user identification and authentication, extended authentication methods, and encryption key management that can provide dynamic, per-station or per-session key determination and rekeying.

- The Wi-Fi Protected Access (WPA): This is used after the 802.1x standard. It is an interim standard adopted by the Wi-Fi Alliance to provide more secure encryption and data integrity while the IEEE 802.11i standard was being ratified. WPA supports authentication through 802.1x (known as WPA Enterprise) or with a preshared key (known as WPA Personal), a new encryption algorithm known as the Temporal Key Integrity Protocol (TKIP), and a new integrity algorithm known as Michael. WPA is a subset of the 802.11i specification. WPA2 is a product certification available through the Wi-Fi Alliance that certifies wireless equipment as being compatible with the 802.11i standard. The goal of WPA2 certification is to support the additional mandatory security features of the 802.11i standard that are not already included for products that support WPA. Like WPA, WPA2 offers both Enterprise and Personal modes of operation.

Because the focus of this thesis is on the authentication part of WLAN, we will emphasize on authentication protocols in the next chapters.

2.1 802.1x

Authentication is the act of ascertaining that a client actually is who it claims to be. If a client passes this test of admission, it is authenticated and is allowed access to the protected resource. To implement this process, a set of tools has been developed. In this section, the authentication mechanism of 802.1x can be found in [3].

IEEE 802.1x is a specification for port-based authentication for wired networks [9]. It has been extended for use in wireless networks. It provides user-based authentication, access control and key transport. 802.1x is designed to be flexible and extensible. It relies on EAP for authentication, which was originally designed for Point-to-Point Protocol (PPP), but was reused in 802.1x. EAP is extensible; hence it can use any authentication mechanism.

Figure 2.1 shows the 802.1x layers.

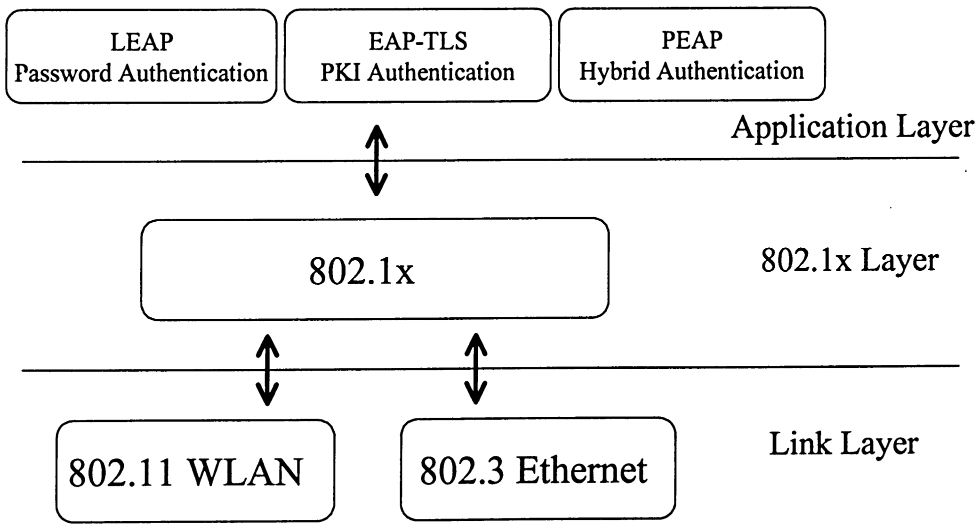


Figure 2.1 802.1x Layers [24]

2.1.1 Three entities in 802.1x

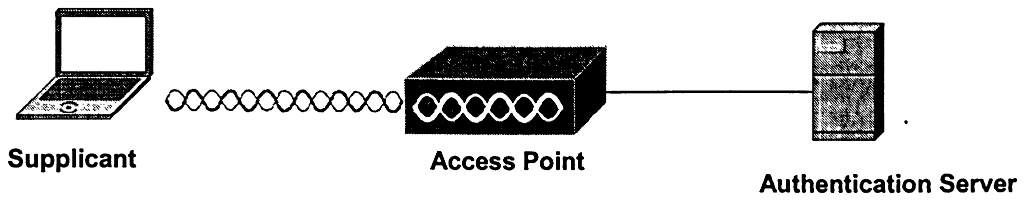
802.1x uses three types of entities: the Supplicant, the Authenticator and the Authentication Server.

- The Supplicant: An entity at one end of a point-to-point LAN segment that is being authenticated by an authenticator attached to the other end of that link. A supplicant entity requests a network access.

- The Authenticator: An entity at one end of a point-to-point LAN segment that facilitates authentication of the entity attached to the other end of that link. An authenticator entity relays 802.1X frames of a client (supplicant) to an authentication server and after the authentication server approves the client's authentication, the authenticator provides network connectivity to the supplicant. The access point may also serves as the Authenticator.

- An Authentication server: an entity that provides authentication service to an authenticator. This service determines from the credentials provided by the supplicant, whether the supplicant is authorized to access the network services provided by the authenticator. An authentication server entity authenticates, authorizes and provides accounting service for clients in the WLAN structure. Typically, the authentication server is a RADIUS server, which is coupled to the wired network authentication.

The three entities are described in Figure 2.2



The supplicant: Resides on the WLAN client
The authenticator: Resides on the access point
The authentication server: Resides on the RADIUS server

Figure 2.2 The three entities in 802.1X.

2.1.2 802.1x is port based

The standard IEEE 802.1x defines a port-based network access control. 802.1x is designed to collect authentication information from supplicant, and grant or deny access based on this information. A port is any kind of controlled access (router, switch, modem line for dial-up, etc). Because 802.1x was not initially designed for wireless networks, to adapt this concept to 802.11, we need to define what is a port in this context. The IEEE community defines a virtual port as an association between a supplicant (station) and an access point. The access control is performed at the MAC level.

A port can be considered as a logical connection between the supplicant and the authenticator. An authenticator supports both the controlled port, which provides network services after the client passes the authentication, and the uncontrolled port, which relays EAP frames between the supplicant and the authentication server. A supplicant first communicates via the uncontrolled port to authenticate. Then, if it is successfully authenticated, the authenticator shifts the supplicant's port from the uncontrolled to controlled state enabling its use of the network services.

Figure 2.3 shows the port based 802.1x.

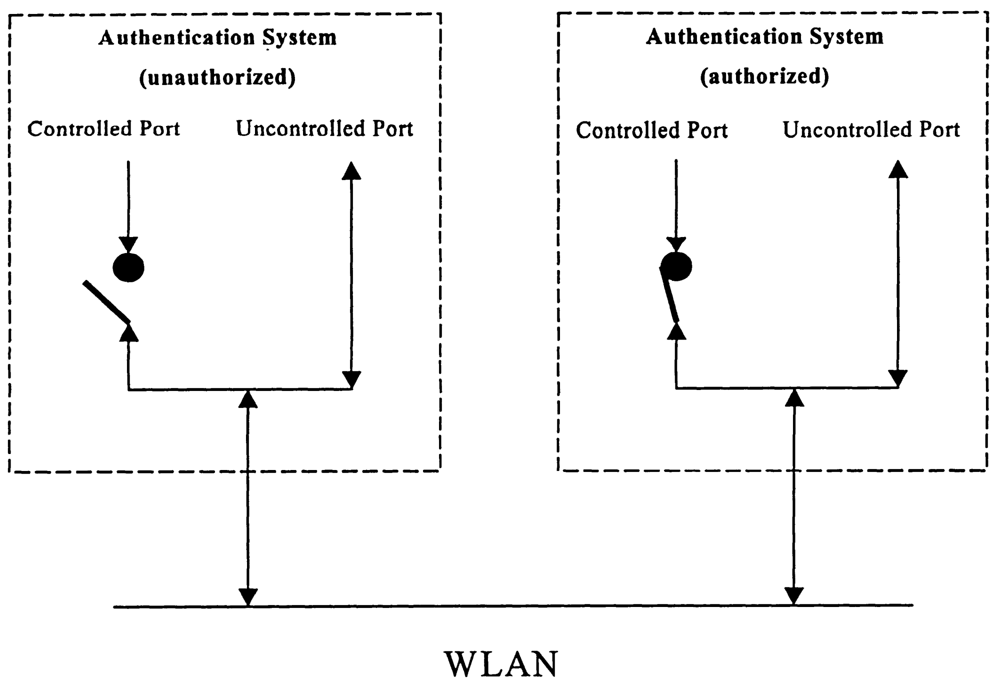


Figure 2.3 The ports in 802.1X [3]

2.1.3 The procedure of 802.1x authentication

When a client device attempts to connect with an access point, the access point responds by enabling a port for passing only EAP traffic from the client to an authentication server (typically a RADIUS server) located on the wired side of the access point. The access point blocks all other traffic, such as HTTP, DHCP, and POP3 packets, until the authentication server can verify the client's identity. Once authenticated, the access point allows other types of traffic on the client's port. In general, the protocol of communication between the client and the access point is EAP over LAN (EAPOL) and the communication protocol between the access point and the authentication server is EAP over RADIUS.

EAP is essentially a transport protocol [4]. Its main advantage is that it provides an authentication framework and can be used by a variety of different authentication types known as EAP methods. It is designed to allow authentication methods to be deployed with no changes to the access point. EAP is used to pass the authentication information between the supplicant and the authentication server. The choice of authentication type is defined by the EAP type. The software supporting the EAP type resides on the authentication server and within the operating system of the client. The access point can be seen as a bridge between the supplicant and the authentication server. One of the goals of EAP is to enable the development of new authentication methods without modifying the access point. One of the key points of 802.1X is that the authenticator can be simple and dumb---all of the brains are in the supplicant and the authentication server. This makes 802.1X ideal for wireless access points, which typically have little memory and processing power. Since the authentication mechanism is independent of the access point, we can specify any EAP method without updating access points [8].

Figure 2.4 shows us details information about the message flow.

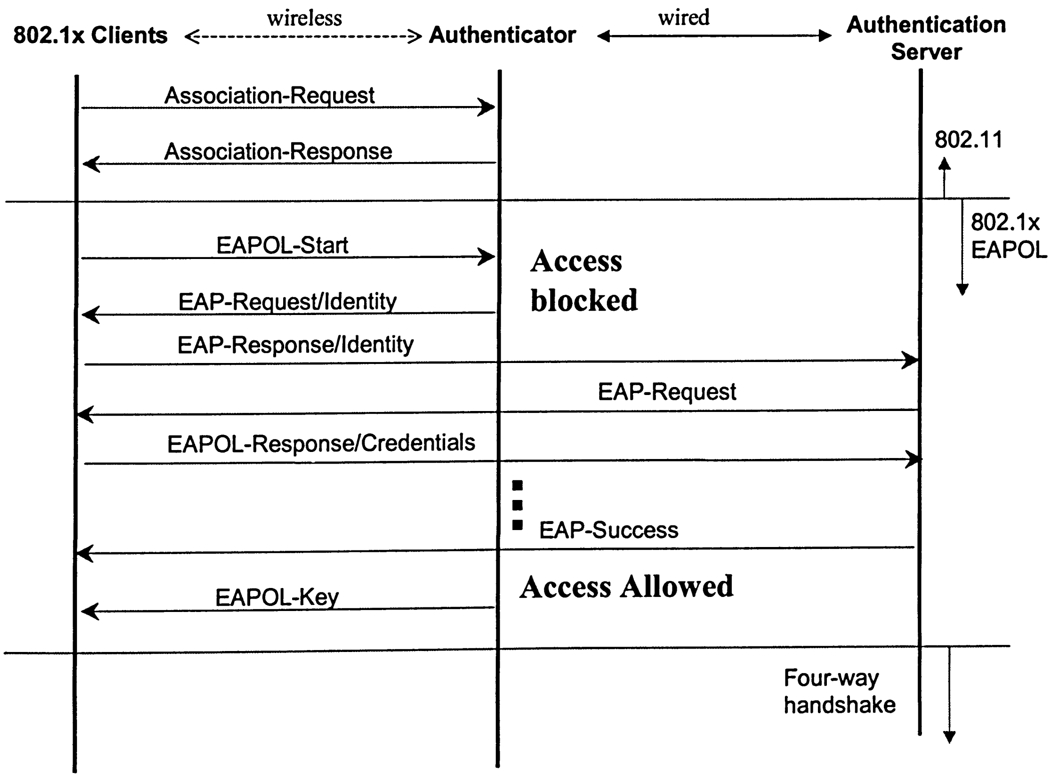


Figure 2.4 802.1x and EAP message flow [7].

There are four message types in EAP protocol: *Request, Respond, Success, Failure*. EAP can encapsulate multiple authentication methods, such as TLS, TTLS and MD5. The Authentication Server uses the Success or Failure message to notify the AP whether the client authentication was successful or not.

The whole message flow is briefly described as follows: The authenticator sends an *EAP-Request/Identity* packet to the supplicant once it has associated with the access point. The *Request-Identity* and *Response-Identity* messages precede other request and response messages. After that, the EAP begins to encapsulate other authentication protocols. If the EAP authentication succeeds, the authentication server and supplicant generate a common secret key called *Master Session Key* (MSK), this key is bind with the MAC address of the supplicant.

Deployments of the IEEE 802.11 wireless LANs today are based on EAP, and use several EAP methods, including EAP-TLS [RFC2716][19], EAP-TTLS, PEAP and EAP-SIM. These methods support authentication credentials that include digital certificates, user-names and passwords, secure tokens, and SIM secrets.

2.2 EAP-TLS

EAP-TLS, EAP Transport Layer Security (TLS) [19], is a Microsoft-supported EAP authentication algorithm based on the TLS protocol (RFC2246)[18]. TLS is the current version of Secure Socket Layer (SSL) used in most Web browsers for secure Web application transactions. TLS has proved to be a secure authentication scheme and is now available as an 802.1X EAP authentication type. EAP-TLS is supported in the Microsoft XP platform, and suppose is planned for legacy Microsoft operating systems as well.

2.2.1 TLS Overview

EAP-TLS is based on SSL v3.0. To better understand EAP-TLS operation, and our modified protocol is also based on TLS, this section focuses on the theory part of TLS. TLS is designed to provide secure authentication and encryption for a TCP/IP connection. To provide this functionality, TLS comprises three protocols [9]:

- The handshake protocol: The handshake protocol negotiates the parameters for the TLS session. The TLS client and server negotiate the protocol version, encryption algorithms, authenticate each another, and derive encryption keys.
- The record protocol: The record protocol facilitates encrypted exchanges between the TLS client and the server. The negotiated encryption scheme and encryption keys are used to provide a secure tunnel for application data between the TLS endpoints.
- The alert protocol: The alert protocol is the mechanism used to notify the TLS client or server of errors as well as session termination.

TLS authentication is generally split into two methods: server-side authentication and client-side authentication. Server-side authentication uses a public key infrastructure (PKI), namely PKI certificates. For TLS authentication, client-side authentication can also use PKI certificates, but this is optional. But for EAP-TLS, it uses both server-side and client-side certificates.

2.2.2 PKI and Digital Certificates

PKI encryption is based on asymmetric encryption keys. A PKI user has two keys: a public key and a private key. Any data encrypted with the public key can be decrypted only with the private key, and vice versa. For example: Bob gives Alice his public key. Alice then sends Bob an e-mail encrypted with his public key. For Bob to read the message, he has to decrypt the message with his private key. Because Bob is the only person with access to his private key, this is the only person that can decrypt the message. The procedure is showed in Figure 2.5.

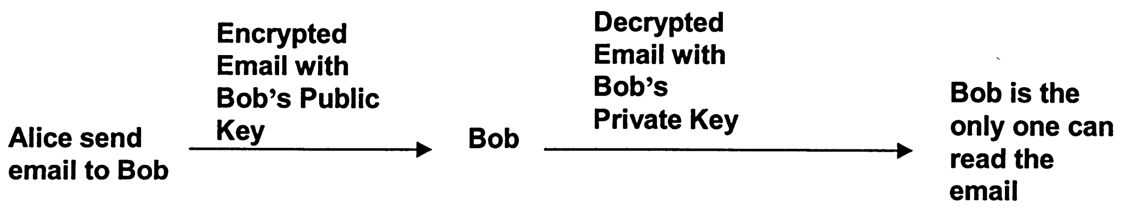

Alice send email to Bob
Encrypted Email with Bob's Public Key
Bob
Decrypted Email with Bob's Private Key
Bob is the only one can read the email

Figure 2.5 A simple PKI flow

Digital certificates are data structures distributed by a certificate authority that join a public key to a user. The digital signature is derived by combining the certificate version, serial number, issuer, user, user's public key, and validity period and running the values through a keyed hash function. The certificate authority keys the hash with its own private key. The procedure is showed in Figure 2.6.

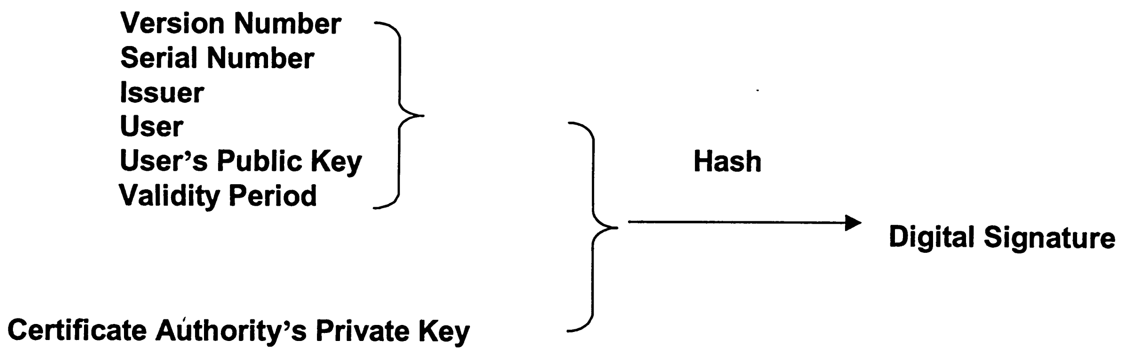


Figure 2.6 The flow of Digital Signature

2.2.3 TLS Authentication Process

The TLS process begins with the handshake process:

1. The TLS client connects to a server and makes an authentication request;
2. The server sends its digital certificate to the client;

3. The client verifies the certificate's validity and digital signature;
4. The server requests the client-side authentication;
5. The client sends its digital certificate to the server;
6. The server verifies the certificate's validity and digital signature;
7. The encryption and message integrity schemes are negotiated;
8. Application data is sent over the encrypted tunnel via the record protocol.

2.2.4 EAP-TLS Authentication Process

EAP-TLS supports mutual authentication and dynamic keying. The server gives a certificate to the client and the client validates it. Once the client is confident of the server's identity, it sends its certificate to the server.

The EAP-TLS authentication process is illustrated as Figure 2.7:

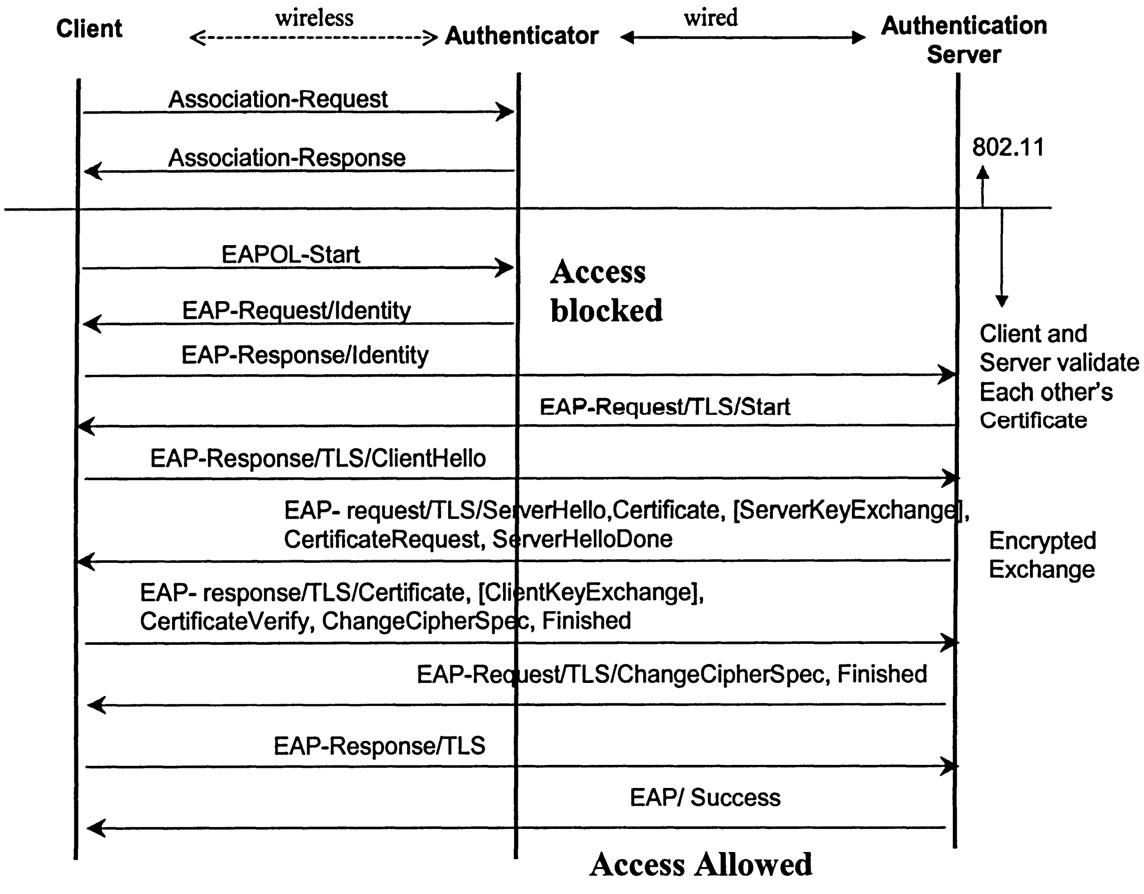

Client
wireless
Authenticator
wired
Authentication Server
Association-Request
Association-Response
802.11
EAPOL-Start
Access blocked
EAP-Request/Identity
EAP-Response/Identity
Client and Server validate Each other's Certificate
EAP-Request/TLS/Start
EAP-Response/TLS/ClientHello
EAP- request/TLS/ServerHello,Certificate, [ServerKeyExchange],
CertificateRequest, ServerHelloDone
Encrypted Exchange
EAP- response/TLS/Certificate, [ClientKeyExchange],
CertificateVerify, ChangeCipherSpec, Finished
EAP-Request/TLS/ChangeCipherSpec, Finished
EAP-Response/TLS
EAP/ Success
Access Allowed

Figure 2.7 EAP-TLS protocol exchange [9]

1. Wireless Client gets associated with the AP.
2. AP does not permit the Client to send any data at this point and the AP sends an Authentication request.
3. The Client will then respond with an EAP- Response Identity with user identity credentials back to the Authentication Server.
4. The Authentication Server such as RADIUS responds back to the Client with an EAP-TLS Start Packet.

The EAP-TLS conversation starts at this point.

5. The Client sends an EAP-Response back to the Authentication Server which contains a Client_hello handshake message, a cipher that is set for NULL (and that will remain this value until change_cipher_spec are negotiated), and TLS

version number.

6. The Authentication Server will present its certificate to the Client as well as request a valid one from the Client. The Authentication Server responds with an EAP-Request packet that contains the following: TLS server_hello, handshake message, server certificate, server_key_exchange, certificate request and server_hello_done.
7. The Client responds with an EAP-Response message that contains the following: Client Certificate, Client_key_exchange, certificate_verify change_cipher_spec and TLS finished Message.
8. After the Client authenticates successfully, the EAP server will respond with an EAP-Request that contains the change_cipher_spec and finished handshake message. The finished handshake message contains the authentication response with the hashed key from the server. Upon receiving this, the Client will verify the hash in order to authenticate the EAP server. A new encryption key is dynamically derived from the master key (The key derived between the EAP client and EAP server during the EAP authentication process i.e. during the TLS handshake).
9. At this point the EAP-TLS protocol enables the port and the wireless Client can access the network.

2.3 EAP-TTLS

Proposed by Funk and Certicom, EAP- Tunneled TLS is an extension of EAP-TLS and provides the benefits of strong encryption without the complexity of mutual certificates on both the client and authentication server. Like TLS, EAP-TTLS supports mutual authentication, and it only requires the authentication server to be validated to the client through a certificate exchange [8].

EAP-TTLS allows the client to authenticate to the authentication server by using usernames and passwords and only requires a certificate for the authentication servers. EAP-TTLS simplifies roll out and maintenance and retains strong security and authentication. A TLS tunnel can be used to protect EAP messages and existing user credential services such as Active Directory, RADIUS, and LDAP can be reused for 802.1X authentication.

EAP-TTLS provides a sequence of attributes that are included in the message. By including a RADIUS EAP-Message attribute in the payload, EAP-TTLS can be made to provide the same functionality as Protected EAP (PEAP) (will discuss in next charter). If, however, a RADIUS password or Challenge Handshake Authentication Protocol (CHAP) password attribute is encapsulated, TTLS can protect the legacy authentication mechanisms of RADIUS. When the TTLS server forwards RADIUS messages to the home server, it encapsulates the attributes by EAP-TTLS and inserts them directly into the forwarded message.

Because this method is similar to PEAP, it is gradually being replaced by PEAP.

Chapter 3. The MEAP method

MEAP is based on PEAP added with SPEKE to ensure the high level authentication. As we know, after typically EAP authentication process, the authentication process leaves two considerations: First, the AP still needs to be authenticated by the client. Second, we still need derive keys to encrypt the traffic. In this thesis, we define MEAP to satisfy mutual authentication and also try to provide simultaneous authentication like EAP-TLS. Also, because MEAP is based on PEAP, MEAP only requires server-side certificates, thus, MEAP is more scalable for large scaled networks.

Figure 3.1 shows the protocol layers for communication between the supplicant and the authenticator.

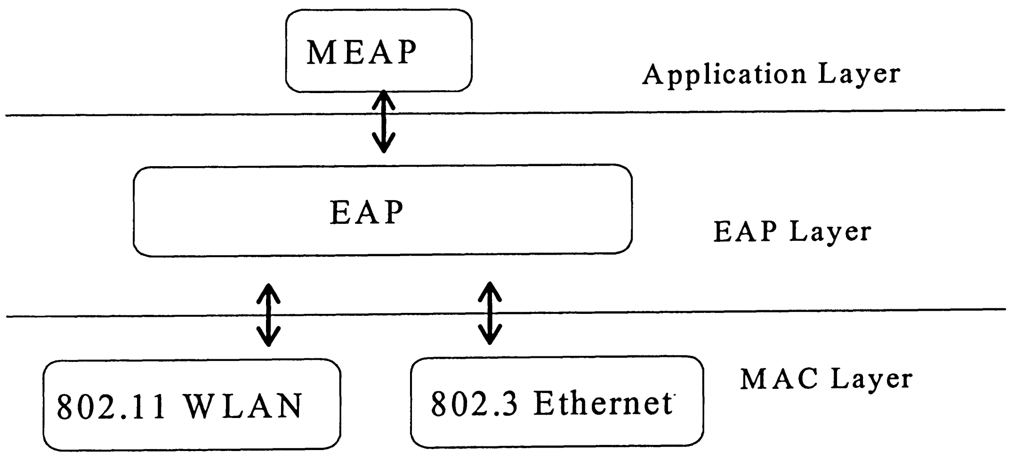


Figure 3.1 EAP Layers

3.1 The problems in EAP protocols

The Wi-Fi Alliance now offers testing for five EAP types for WPA/WPA2 certification: EAP-TLS (part of the original WPA test suite, although formally required, just de facto), EAP-TTLS/Microsoft CHAPv2 (MSCHAP) (common TTLS method), PEAPv0

/EAP-MSCHAPv2 (Microsoft's version), PEAPv1/EAP-GTC (Cisco's version), and EAP-SIM (of wide interest to cell operators for Wi-Fi cell authentication convergence) [21].

All of these EAP methods still have certain amounts of risk in part because of the potential for authentication information to be wormed out of less secure systems, and then broken through brute force offline. William A. Arbaugh [27] maintains a list of currently well-known problems; it is a constantly updated site.

EAP is not a secure protocol: it sends its messages in the clear. A method of creating an encrypted EAP session using TLS appeared, and was shipped by Microsoft and others as EAP-TLS, but it requires an installation of client certificate on every computer that wants to connect. It also leaves some useful information in the clear, although it is seemingly impossible to exploit that information.

EAP-TLS offers mutual authentication, however, in which the client and authentication server can verify each other's identity before the start of the transaction.

Two fixes to the EAP-TLS's problem appeared in the form of EAP-TTLS and PEAP. Both methods first start a TLS session using a server-side certificate, and then pass user authentication using an inner method for the actual user credentials.

PEAP's success appears to be a given as time continues to pass and more companies give up legacy authentication in favor of EAP methods which would be fully supported under PEAP, and as third parties continue to support the encrypted standard. The currently most common used inner method of PEAP is MSCHAP. Microsoft has shipped PEAP updates to Windows XP and 2000 [11].

But PEAP plus MSCHAP limited PEAP to only running on Windows-based Operation System, if users want to using other Operation system, PEAP can't work properly, and may need the vendor to develop new firmware to support it. So PEAP plus SPEKE is developed in this thesis.

3.2 PEAP

Protected EAP is a draft EAP authentication type that is designed to allow hybrid authentication [5]. PEAP employs server-side PKI authentication. For client-side authentication, PEAP can use any other EAP authentication type. Because PEAP establishes a secure tunnel via server-side authentication, non-mutually authenticating EAP types can be used for client-side authentication, such as EAP generic token card (GTC) for one-time passwords (OTP), and EAP MD5 for password based authentication.

PEAP is based on server-side EAP-TLS, and it addresses the manageability and scalability shortcomings of EAP-TLS. Organizations can avoid the issues associated with installing digital certificates on every client machine as required by EAP-TLS and select the method of client authentication that best suits them.

3.2.1 PEAP Authentication Process

PEAP protocol has two phases. The first phase is to establish a secure tunnel using the EAP-TLS with server authentication. The second phase implements the client authentication based on EAP methods, the exchange of other PEAP-specific capabilities through the secure transport established during phase 1.

PEAP authentication begins in the same way as EAP-TLS:

1. The client sends an EAP Start message to the access point;
2. The access point replies with an EAP Request Identity message;
3. The client sends its Network Access Identifier (NAI), which is its username, to the access point in an EAP Response message;
4. The access point forwards the NAI to the RADIUS server encapsulated in a RADIUS Access Request message;
5. The RADIUS server will respond to the client with its digital certificate;
6. The client will validate the RADIUS server's digital certificate;

From this point on, the authentication process diverges from EAP-TLS;

7. The client and server negotiate and create an encrypted tunnel;
8. This tunnel provides a secure data path for client authentication;
9. Using the TLS Record protocol, a new EAP authentication is initiated by the RADIUS server;
10. The exchange will include the transactions specific to the EAP type used for client authentication;
11. The RADIUS server sends the access point a RADIUS ACCEPT message, including the client's WEP key, indicating successful authentication.

3.2.2 Common PEAP Authentication weakness

The current common PEAP are always using Microsoft PEAP, which supports client authentication by only MSCHAP Version 2. Microsoft PEAP limits user databases to those that support MSCHAP Version 2, such as Windows NT Domains and Active Directory [22].

Also, to use Microsoft's PEAP, users must purchase individual certificates from a third-party Certification Authority (CA) to install on their Internet Authentication Service

(IAS), and a certificate must be installed in the user's local computer certificate store. For wireless clients to validate the IAS certificate chain properly, the root CA certificate must be installed on each wireless client.

Windows XP, however, includes the root certificates of many third-party CAs. If the IAS server certificates correspond to an included root CA certificate, no additional wirelss client configuration is required. If users purchase IAS server certificates for which Windows XP does not include a corresponding root CA certificate, they must install the root CA certificate on each wireless client.

The install procedure of Microsoft's PEAP is more likely with the installation of EAP-TLS, which also need install client certification on client. The benefit of PEAP is wasted, or more likely to say, denied in this certificate installation process. In response to the inconvenience of Microsoft PEAP method, SPEKE is chosen in this thesis.

3.3 The SPEKE used in MEAP

The SPEKE relies on exponentiation involving large random numbers modulo a large prime number. The exponentiation operator can be considered a one-way function due to the difficulty of calculating discreet logarithms (the inverse of exponentiation). Each party – the user station and the authenticator – will work from its knowledge of the user's password p to calculate a common master session key K. To do this, each party generates a large random number. The user station generates the value a, and the authenticator generates the value b. Each party only knows one value; no one ever knows both. The password p, which is known to both the user and the authenticator, is small and easy to remember [6].

Initially, the AS and the client share the values:

m= large prime number

p: user's password

Let define the following terms used in MEAP:

a: a large random value generated by the Client

b: a large random value generated by the Authenticator

| : the concatenation operator

K: Master session key, is independently calculated by each party

In the discussion below, we consider two parties, the client and the authenticator. In wireless networks, the authenticator is the AP. In practice, however, there is often a third party – a backend authentication server that the AP consults. In this case, it is the authentication server that actually plays the role of authenticator in the discussion below with the AP acting in a pass-through role. A RADIUS server is using as authentication server and AP is omitted in this thesis.

The following figure 3.2 describes the flow of SPEKE:

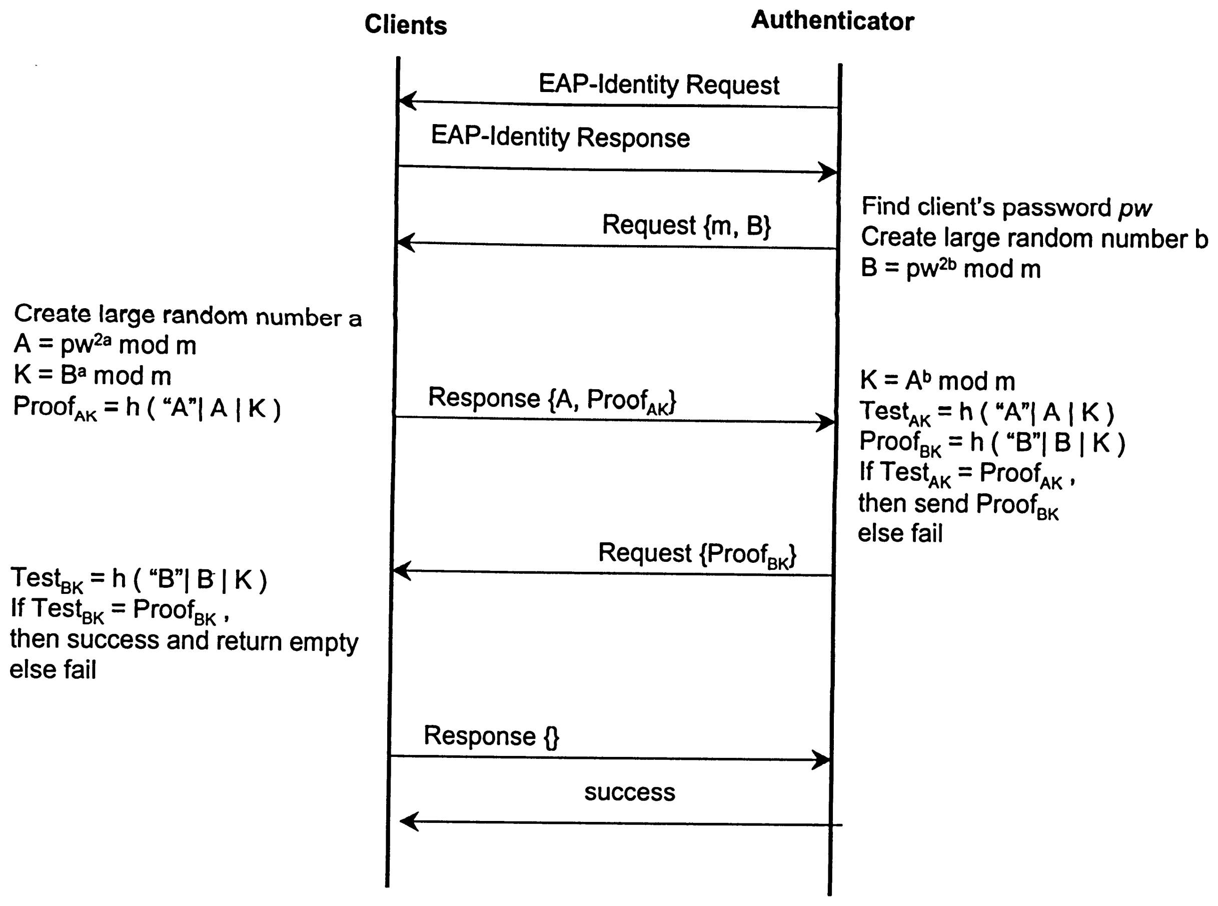


Figure 3.2 the flow of SPEKE [6]

The authenticator kicks off the EAP conversation by sending an EAP-Identity Request to the user station. The client replies with the user's identity.

When the authenticator receives the client's EAP-Identity Response, it looks up the user in its access repository and retrieves the client's password *p*. Next, the authenticator creates a large random number *b* and calculates

$$B = p^{2b} \bmod m$$

Where *B* is an intermediate value and *m* is a large prime number used as the modulus. The authenticator sends *m* and *B* to the user station in an EAP-SPEKE Request message.

After receive the request, the client creates another large random number "*a*" and calculates

$$A = p^{2a} \bmod m$$

Next the client calculates

$$K = B^{a} \bmod m$$

where *K* is the user station's calculation of the master session key and *B* is the value received from the authenticator. Finally client calculates

$$\text{Proof}_{AK} = h\,(\text{“A”} \mid A \mid K)$$

where *Proof$_{AK}$* is the proof that *A* knows *K*, h is a secure, one-way hash function, “A” is the ASCII string containing a capital A.

The client now sends an EAP response to the Authenticator containing A and Proof$_{AK}$. When the authenticator receives the response to its first EAP-SPEKE Request, it calculates

$$K = A^{b} \bmod m$$

where *K* is the authenticator's calculation of the master session key and *A* is the value received from the user. Next the authenticator calculates

$$\text{Test}_{AK} = h\,(\text{“A”} \mid A \mid K)$$

and

$$\text{Proof}_{BK} = h\,(\text{“B”} \mid B \mid K)$$

Now the authenticator compares Test$_{AK}$ to the value Proof$_{AK}$ received from the client. If they are not equal, the authenticator signals failure. If they are equal, the authenticator sends a second EAP-SPEKE Request to the user station.

When the user station receives the second EAP-SPEKE Request, it calculates

$$\text{Test}_{BK} = h\,(\text{“B”} \mid B \mid K)$$

from values received or calculated earlier. The user station compares Test$_{BK}$ to the value Proof$_{BK}$ received from the authenticator. If they are not equal, the user station aborts the attempted session. If they are equal, the user station returns an empty EAP-SPEKE

Response to the authenticator to signal that it is satisfied with the authentication.

When the authenticator receives the empty response, it returns an EAP Success message to the user station as a final signal that the authentication succeeded.

The session key *K* is independently calculated by each party. This works due to the associative property of exponentiation. It is computationally infeasible for an attacker to work backward from the values *A* or *B* to calculate *p* due to the difficulty of calculating discrete logarithms, the inverse function to exponentiation. Note also that if *p* is compromised, an attacker listening in on an authentication between the user and the authenticator is still unable to calculate *K*. To calculate *K* he would need to know the values of both *a* and *b* which are very large random numbers. Neither can the attacker work backward from $Proof_{AK}$ or $Proof_{BK}$ to calculate *K* because h is a one-way hash function. This inability to calculate *K* even if *p* is known is what gives SPEKE the property of forward secrecy defined.

The master session key *K* itself is not used as a WEP or TKIP key to encrypt the wireless data session being established. Those keys are derived from *K* using a key derivation function.

3.4 The procedure of MEAP

The message flow is shown on figure 3.3-1 and 3.3-2.

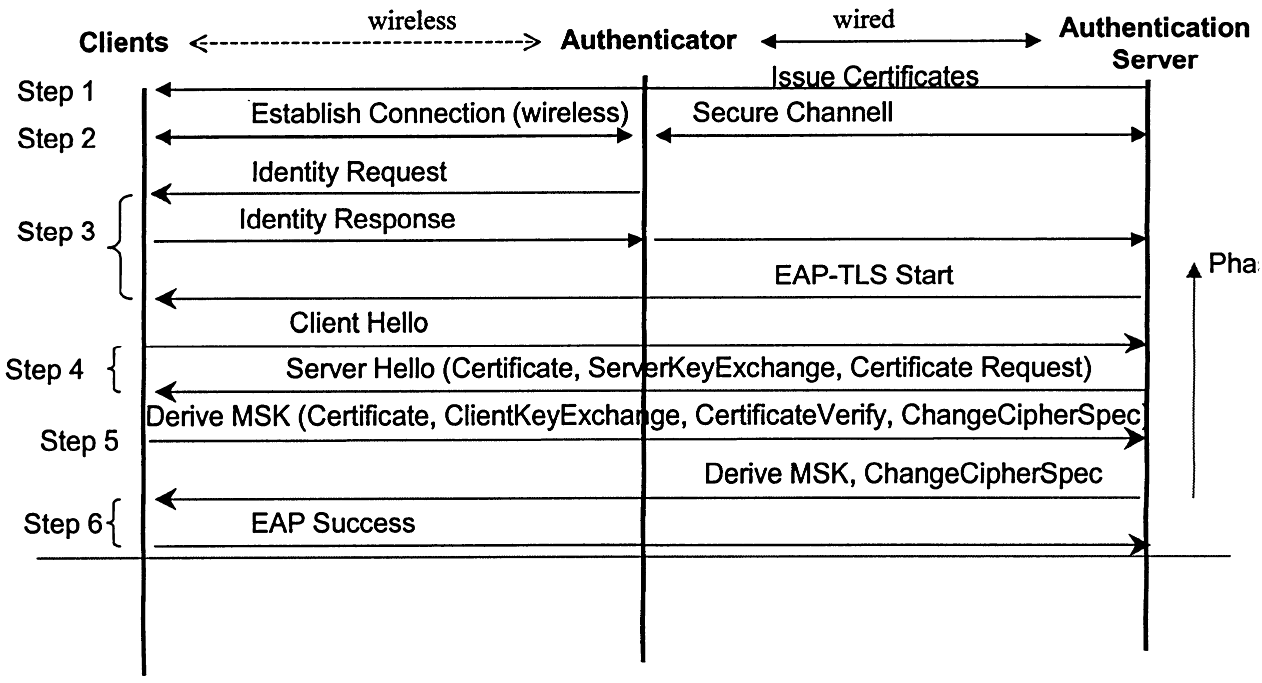

Clients
wireless
Authenticator
wired
Authentication Server
Step 1
Issue Certificates
Step 2
Establish Connection (wireless)
Secure Channell
Identity Request
Step 3
Identity Response
EAP-TLS Start
Pha
Client Hello
Step 4
Server Hello (Certificate, ServerKeyExchange, Certificate Request)
Derive MSK (Certificate, ClientKeyExchange, CertificateVerify, ChangeCipherSpec)
Step 5
Derive MSK, ChangeCipherSpec
Step 6
EAP Success

Figure 3.3-1 The message flow of MEAP phase1

Step 1

Similar to EAP-TLS, the EAP server requires a certificate; the client/peer certificate is optional.

Step 2

The client/peer must establish a connection to the authenticator—in this case, a wireless connection. An important requirement is the secure channel between the authenticator and the EAP server. This is vital because the specification does not indicate how this is established, but it requires one.

Step 3

The identity request-response is the basic EAP sequence, which is sent in the clear. In PEAP, this is used for administrative purposes, such as which server to select, and possibly for other initial context setup. The identity, which is sent in the clear, should not be used for any other purposes. Any identity exchange should happen in phase 2 after the secure tunnel is established—for example, tunneling the identity request-response using the EAP-TLV. EAP-TLV is a payload with standard Type-Length-Value (TLV) objects. The identity response is sent to the EAP server, which in turn starts the process with the EAP-TLS start message.

Steps 4, 5, and 6

These steps are typical EAP-TLS exchanges. Usually the client certificate is not exchanged. The successful completion of the EAP-TLS ends phase 1, and phase 2 leverages the secure tunnel created by phase 1.

From step 7, the phase 2 began.

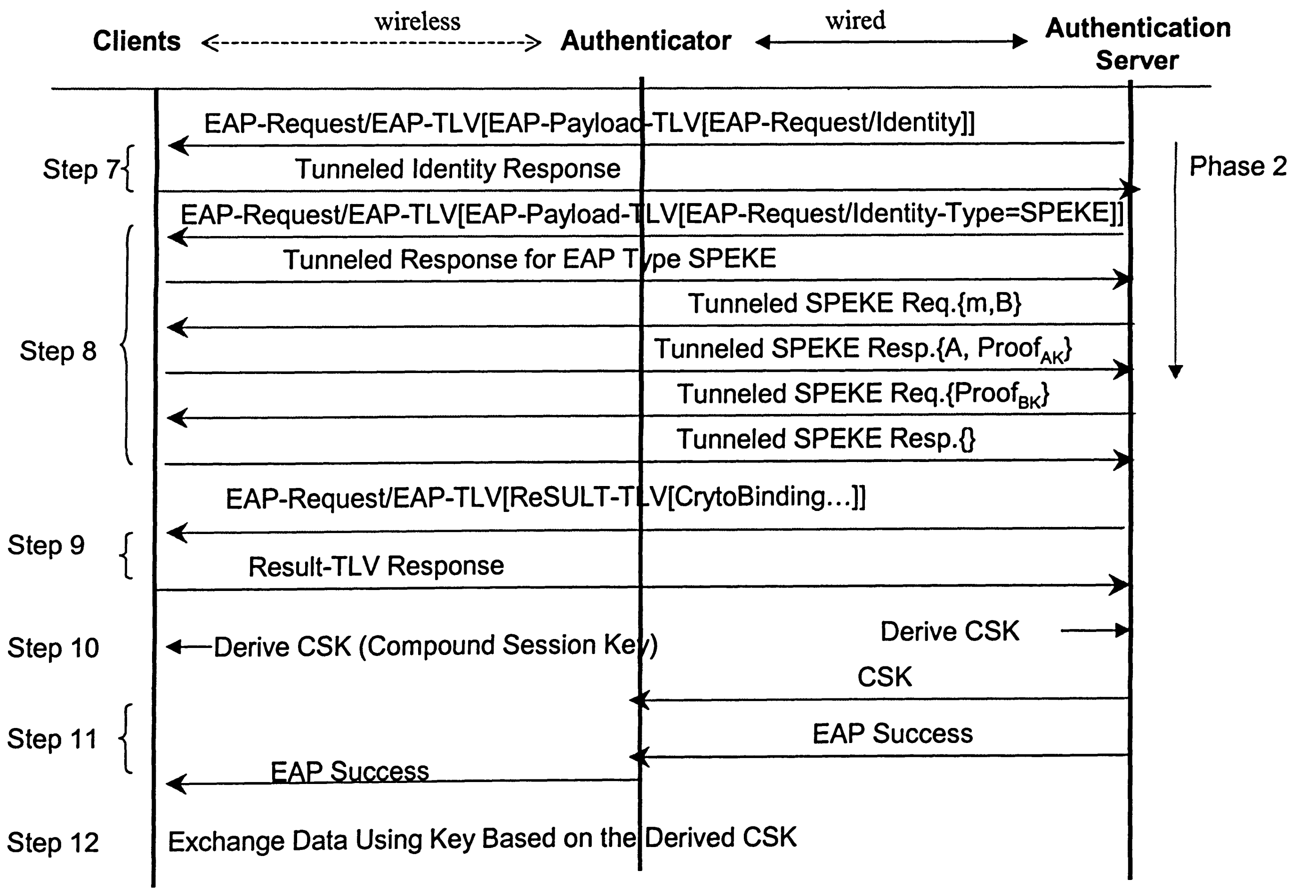

Clients
wireless
Authenticator
wired
Authentication Server
EAP-Request/EAP-TLV[EAP-Payload-TLV[EAP-Request/Identity]]
Step 7
Tunneled Identity Response
Phase 2
EAP-Request/EAP-TLV[EAP-Payload-TLV[EAP-Request/Identity-Type=SPEKE]]
Tunneled Response for EAP Type SPEKE
Tunneled SPEKE Req.{m,B}
Step 8
Tunneled SPEKE Resp.{A, Proof_AK}
Tunneled SPEKE Req.{Proof_BK}
Tunneled SPEKE Resp.{}
EAP-Request/EAP-TLV[ReSULT-TLV[CrytoBinding...]]
Step 9
Result-TLV Response
Step 10
Derive CSK (Compound Session Key)
Derive CSK
CSK
Step 11
EAP Success
EAP Success
Step 12
Exchange Data Using Key Based on the Derived CSK

Figure 3.3-2 The message flow of MEAP phase 2

Step 7

This is the beginning of phase 2. The EAP-TLV mechanism can be used to tunnel the normal EAP identity exchange.

Step 8

In this step, the EAP server authenticates the client using any of the EAP mechanisms: EAP-MD5, EAP-CHAP, EAP-SIM, and so on. Typically, is using MSCHAPv2. But MSCHAPv2 is only support windows operation system, so SPEKE is used in here. The exchange is fully protected by the TLS tunnel, and the EAP-TLV choreography allows a graceful mechanism to affect the EAP mechanisms. This is the heart of the PEAP method—the server with a certificate, the establishment of the tunnel by TLS, and the use of the EAP methods available in the organization's infrastructure.

Step 9

This is the final stage of crypto binding and so on between the client and the EAP server.

Step 10

In this step, the client and the server derive the required keys.

Step 11

This is where the authenticator receives the keys and the result of the authentication process.

Step 12

Now the client and AP can exchange information using the keys that are derived from the PEAP mechanism.

Chapter 4. MEAP Implementation

We implement the MEAP by add SPEKE part on FreeRADIUS by C and running it on Linux Redhat 9.0. The implementation shows that the client and authenticator can authenticate each other.

4.1 Remote Authentication Dial-in User Service (RADIUS)

To implement MEAP, the RADIUS server should be first introduced. The main server side code is based on RADIUS. The clearly understand of RADIUS will help us understand the whole procedure.

The RADIUS protocol is based on a client/server model. A network access server (NAS), which in WLAN is the wireless Access Point, operates as a client of RADIUS. It is responsible for passing user's information to a designated RADIUS server and then acting on the response that is returned. In this testing, I set the NAS's address to 127.0.0.1, which supposes the RADIUS server and AP existing on one PC.

All RADIUS messages are sent as User Datagram Protocol (UDP) messages. Only one RADIUS message is included in the UDP payload of a RADIUS packet.

A RADIUS packet (Figure 4.1) consists of a RADIUS header and RADIUS attributes. Each RADIUS attribute specifies a piece of information about the connection attempt. For example, there are RADIUS attributes for the user name, the user password, the type of service requested by the user, and the IP address of the NAS (access point). RADIUS attributes are used to convey information between RADIUS Clients (access point) and RADIUS servers.

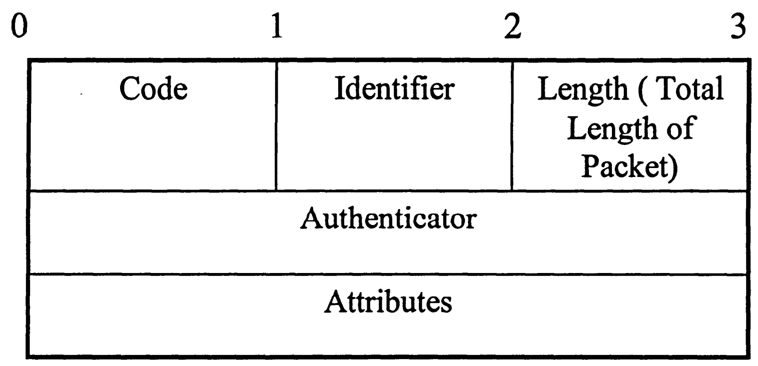


FIGURE 4.1 RADIUS PACKET FORMAT

Code (1 Byte), identifies the type of RADIUS packet.

1: Access-Request

2: Access-Accept

3: Access-Reject

4: Accounting-Request

5: Accounting-Response

11: Access-Challenge

12: Status-Server (experimental)

13: Status-Client (experimental)

255: Reserved

Identifier: This helps 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.

Length: This value indicates the total length of the packet. The ideal length of the packet should be between 20 to 4096.

Authenticator: This can represent a Request or a Response. The Request Authenticator value is a 2-byte random number with a unique and unpredictable value. The Response Authenticator is a response challenge with a combination of Code + ID + Length + RequestAuth + Attributes + Secret.

Then Let's take a look at RADIUS packet type [29][30].

- Access-Request: Sent by a RADIUS Client to request authentication and authorization for a network access connection attempt. It determines whether a user is allowed access to a specific NAS, and any other specific service.
- Access-Accept: Sent by a RADIUS server in response to an Access-Request message when all conditions are met. The message informs the RADIUS Client that the connection attempt is authenticated and authorized and it contains the list of configuration values for the user.
- Access-Reject: Sent by a RADIUS server in response to an Access-Request message if any condition is not met. This message informs the RADIUS Client that the connection attempt is rejected. A RADIUS server sends this message if either the credentials are not authentic or the connection attempt is not authorized.
- Access-Challenge: Sent by a RADIUS server in response to an Access-Request message if all conditions are met and RADIUS server wishes to issue a challenge to which the user must respond. The Client in response resubmits its original Access-Request with a new request ID, response (encrypted), and including the Attribute from the Access-challenge.
- Accounting-Request: Sent by a RADIUS Client to specify accounting information for a connection that was accepted.
- Accounting-Response: Sent by the RADIUS server in response to the Accounting-Request message. This message acknowledges the successful receipt and processing of the Accounting-Request message.

So how does RADIUS server work with EAP protocol? The RADIUS server encapsulates the EAP packets within a standard RADIUS packet using the EAP-Message attribute, and then transmits them back and forth between the RADIUS Client and the RADIUS server [12]. The access point then becomes only a pass-through device relaying EAP authentication messages from the wireless Client device to the RADIUS server and vice versa without having to process the packets. In this way, every new EAP type that is introduced need not be installed at each access point but only at the RADIUS server end. The access point would then behave the same way for each EAP type thus making it extremely flexible.

The RADIUS Client (access point) must however support the negotiation of EAP as an authentication protocol and the passing of EAP messages to a RADIUS server. When a connection attempt is made, the end user negotiates the use of EAP with the access point. When the user sends an EAP message to the access point, the access point encapsulates the EAP message as a RADIUS message and sends it to its configured RADIUS server. The RADIUS server processes the EAP message and sends a RADIUS-formatted EAP message back to the access point. The access point finally forwards the EAP message to the end user.

4.2 The main structure of MEAP

MEAP in fact is the combination of PEAP and SPEKE. But how can these two different authentication protocols work together? Here we should also know two main features of EAP.

The first feature is that, EAP separates the message exchange from the process of authentication by providing an independent exchange layer. Because of this property, EAP can be easily extended to support the other security features.

The other feature is orthogonal extensibility, meaning that the authentication processes can extend the functionality by adopting a newer mechanism without necessarily effecting a corresponding change in the EAP layer.

Based on this feature, PEAP can work together with SPEKE.

4.2.1 The EAP packets

The format of EAP packets is illustrated in Figure4.2 [4]. All EAP packet headers begin with four octets header. First eight bits indicate the code of the message. The next eight bits are the Identifier that helps in matching Responses with Requests. The rest 16 bits describe the Length of the EAP packet in octets including the header. This is also the format of the EAP-Success and the EAP-Failure messages, which have no data to be carried [28].

0	1	2	4
Code	Identifier	Length (Total Length of Packet)	Data •••

Figure 4.2 EAP packet format [4]

Code (1bytes)

1: Request

2: Response

3: Success

4: Failure

If the code = 1 or 2, then EAP-Request and EAP-Response message is define in Figure 4.2. After the common EAP header, there is a 1 bytes value Type, which defines the type of the Request or the Response. Normally the Type of the Response is the same as the Type of the Request it is replying. The Type-Data field carries the actual payload of the Request of the Response and its length and contents depend on the Type of the particular message.

0 1 2 4

Code	Identifier	Length (Total Length of Packet)
Type	Type-Data •••	

Figure 4.3 EAP Request and EAP Response packet format [4]

All the EAP implementation should base on the EAP packet format. In this program, eapcommon.c defines the common code to clients and server. Here I add the eap_type = speke in it to let server knew that EAP type = speke.

4.3 The structure of MEAP modules

4.3.1 The structure of the complete RADIUS Modules

FreeRADIUS is an authentication server. The server performs the functionalities of receiving a RADIUS request, processing the request, and responding to the request.

The FreeRADIUS code is organized in a modular format. It handles all RLM (RADIUS Load Module) requests through a module interface. All modules are present in the local sub-directory "src/modules/" of the installed main source code directory.

FreeRADIUS implements EAP as a module. All EAP-Types are organized as sub-directories in "src/modules/rlm_eap/types/" directory and are handled in their respective sub modules. Each EAP-Type contains a block of code that knows how to deal with a particular kind of authentication mechanism. Thus RLMs first processes a request of the particular authentication type say EAP in "src/modules/rlm_eap/eap.c". This file would contain the basic EAP and generalized interfaces to all the EAP-Types. From here based on the EAP type specified, further authentication is performed via the "rlm_eap_XXX" module where XXX is an EAP_Type name defined in the module section of the RADIUSd.conf file.

The used RADIUS source code directory is shown as figure 4.4.

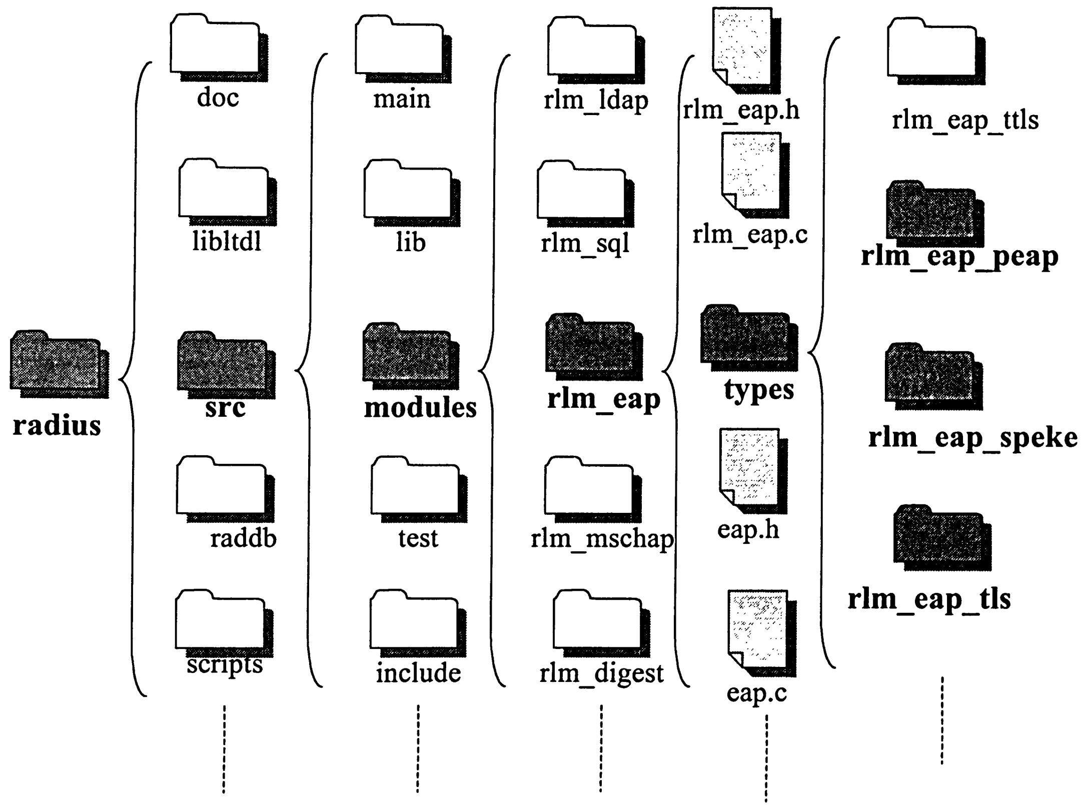


FIGURE 4.4 FREERADIUS CODE STRUCTURE

We created the folder named rlm_eap_speke under the folder FreeRADIUS-1.1.2/src/modules/rlm_eap/types. All the SPEKE part codes are stored and execute from here.

The whole program uses the main module EAP, and sub-module PEAP and EAP_TLS. Another sub-module SPEKE is created. Here, the orthogonal extensibility feature of EAP is used.

4.3.2 The structure of EAP modules

The rlm_eap module deals with EAP authentication mechanisms and the virtual interface to interact with all the EAP-Types. One of the most important function, authenticate( ) is used among all sub modules.

> *int authenticate(void **type_arg, EAP_HANDLER *handler)* This uses specific EAP-Type authentication mechanism to authenticate the user. During authentication, many EAP-Requests and EAP-Response take place for each authentication. Hence authenticate() function may be called many times. EAP_HANDLER contains the complete state information required.

4.3.2.1 eap.h

First, let's take a look at **eap.h,** which define all the important functions that used in all sub modules.

- EAP_DS contains all the received/sending information, where

  response = Received EAP packet

  request = Sending EAP packet

```
typedef struct eap_ds {
        EAP_PACKET        *response;
        EAP_PACKET        *request;
        int             set_request_id;
} EAP_DS;
```

- EAP_HANDLER is the interface for any EAP-Type. Each handler contains information for one specific EAP-Type. This way we don't need to change any interfaces in future.

```
typedef struct _eap_handler {
        struct _eap_handler *next;
        uint8_t         state[EAP_STATE_LEN];
        uint32_t        src_ipaddr;
        unsigned int    eap_id;
        unsigned int    eap_type;
        time_t          timestamp;
        REQUEST         *request;
        char            *identity; /* User name from EAP-Identity */
        EAP_DS          *prev_eapds;
        EAP_DS          *eap_ds;
        void            *opaque;
        void            (*free_opaque)(void *opaque);
        int             status;
        int             stage;
} EAP_HANDLER;
```

where ,

eap_id = copy of the eap packet we sent to the

next = pointer to next

state = state attribute from the reply we sent

state_len = length of data in the state attribute.

src_ipaddr = Client which sent us the RADIUS request containing this EAP conversation.

eap_id = copy of EAP id we sent to the Client.

timestamp = timestamp when this handler was last used.

identity = Identity, as obtained, from EAP-Identity response.

request = RADIUS request data structure

prev_eapds = Previous EAP request, for which eap_ds contains the response.

eap_ds = Current EAP response.

opaque = EAP-Type holds some data that corresponds to the current EAP-request/response

free_opaque = To release memory held by opaque, when this handler is timedout & needs to be deleted. It is the responsibility of the specific EAP-TYPE to avoid any memory leaks in opaque. Hence this pointer should be provided by the EAP-Type if opaque is not NULL

status = finished/onhold/..

- eap_type_t, define the Interface to call EAP sub modules:

```
typedef struct eap_type_t {
        const   char *name;
        int     (*attach)(CONF_SECTION *conf, void **type_data);
        int     (*initiate)(void *type_data, EAP_HANDLER *handler);
        int     (*authorize)(void *type_data, EAP_HANDLER *handler);
        int     (*authenticate)(void *type_data, EAP_HANDLER *handler);
        int     (*detach)(void *type_data);
} EAP_TYPE;
```

4.3.2.2 eap_type.h

The EAP packet structure is define in eap_type.h

- Eaptype_t define EAP-Type specific data:

```
typedef struct eaptype_t {
        unsigned char  type;
        unsigned int   length;
        unsigned char  *data;
} eaptype_t;
```

- eap_packet define the structure to hold EAP data:

```
typedef struct eap_packet {
        unsigned char  code;
        unsigned char  id;
        unsigned int   length;
        eaptype_t      type; //EAP-Type specific Data
        unsigned char     *packet;
} EAP_PACKET;
```

where length = code + id + length + type + type.data

= 1 + 1 + 2 + 1 + X (bytes)

4.3.3 The structure of MEAP module

The implementation of MEAP module is not easy. Because PEAP, EAP-TLS, SPEKE sub-modules are all called for the authentication. PEAP and EAP-TLS are already implemented by FreeRADIUS, so I will emphasize on SPEKE part. But go through the PEAP part is still useful to understand the MEAP module.

First, PEAP module should be defined. The module instance is created by adding the module name inside the modules{} block in the RADIUSd.conf file. The instance definition as placed in the RADIUSd.conf file and the corresponding structure as defined in the header file of the module is as follows:

```
## RADIUSd.conf -- FreeRADIUS server configuration file.
modules {
        #   For all EAP related authentications
        eap {
                        default_eap_type = peap ....
                        peap {
                                default_eap_type = speke
                                  }...
                        }...
        }
```

The rlm_eap_peap module, as described in the rlm_eap module section, deals with the standard attach, detach, and authenticate interfaces. It is to be noted that unlike in the rlm_eap and rlm_eap_tls modules, the rlm_eap_peap module would not have an initiate() interface. This is because PEAP is a protocol on top of TLS, so before initiating the PEAP we have to initialize the TLS session and hence the main EAP module should be taking care of calling the eaptls_initiate(). In other words, eap_tls module should also be called. Thus the final rlm_eap_peap module instantiate structure looks like this:

```
/* rlm_eap_peap.c   - Contains interfaces called from the main module EAP   */
EAP_TYPE rlm_eap_peap = {
        "eap_peap",             /* module_name */
        eappeap_attach,                 /* attach */
        NULL,                   /* No peap initialization interface*/
        NULL,                  /* No need for authorization interface*/
        eappeap_authenticate,/* authentication */
        eappeap_detach                 /* detach */
    };
```

4.3.4 The structure of SPEKE module

At first, the structure of SPEKE module has to be created.

There are three main files used in SPEKE module.

- Eap_speke.h: The header file of SPEKE submodule.

- Eap_speke.c: EAP SPEKE functionality. This program provide the basic SPEKE function, the encryption is implement here. SHA-160 is choosing as the hash method.

- Rlm_eap_speke.c: Handles that are called from EAP. This program contains the main authentication steps.

Under this module, the client-side program eapclient.c is also stored here. It is not mandated to store here, but just a location for this file. Spekeclient.c: EAP specific RADIUS packet debug tool. This program is modified based on the radeapclient.c that provide by FreeRADIUS. It sends arbitrary RADIUS packets to a RADIUS server, and then shows the reply. It can be used to monitor the exchange of the message flow between server and client.

4.3.5 The coding description of SPEKE module

In this section, the details of every program are described.

4.3.5.1 eap_speke.c

eap_speke.c defines the eap-SPEKE main functionality. As I described in Chapter3, SPEKE is more mathematic related, another MIRACL library is chosen to implement SPEKE.

One most important feature of SPEKE is using hash function to encrypt the user's information. Here the MIRACL library is used to do the hash and other mathematic related functions in SPEKE.

MIRACL is a big number library, which implements all of the primitives necessary to design Big Number Cryptography into the real-world application. It is primarily a tool for cryptographic system implementers. MIRACL allows work directly and efficiently with the big numbers that are the building blocks of number-theoretic cryptography. MIRACL now provides more support for conventional cryptography. The latest version implements the new AES, Modes of Operation, and the new hashing standards SHA-160/256/384/512.

The Version 5.00 is chosen because of its support for very constrained environments.

The main functions of eap_speke.c are described as following:

1) At first we should define a prime number. Please note, not a random selected number can be the prime number, so we use the genprime( ) function in MICACL

to generate a real prime number.

2) hash( ), is to hash the character string to 160-bit big number.

3) mkstring( ), is concatenated the characters to a string.

4) SPEKE_HANDLER is to allocate a new SPEKE_HANDLER

```
{ SPEKE_HANDLER *rp
    a. rp-> prime          # that's the large random prime number
    b. rp-> random               # random number value
    c. rp-> intermediate              # own proof value
    d. rp-> peer                 #peer's proof value
    e. rp-> key                  #K, master seesion key
}
```

5) speke_proof( ), is to caculate Proof value.

   $\mathrm{Proof}_{AK} = h(\text{"A"} \mid A \mid K)$,

   $\mathrm{Test}_{AK} = h(\text{"A"} \mid A \mid K)$,

   $\mathrm{Proof}_{BK} = h(\text{"B"} \mid B \mid K)$

   The above value is calculated by this function

6) speke_big2tlv ( ), this function is change the big number to ASCII type. It is to avoid the error in the calculation of big numbers.

4.3.5.2 rlm_eap_speke.c

This program is the main authentication part, and runs at server side.

The main are described as following:

1) speke_initiate ( ), is the function after receive peer's eap indentity.

```
{
  Allocate a SPEKE HANDLER,
  SPEKE_HANDLER* speke = eapspeke_alloc( )
  Compose the speke packet from the data structure, and prepare to send to the
peer.
   Len = speke_big2tlv ( )
  Copy the len to EAP message
   memcpy( )
  Set EAP message's content and length, and set the coding mode to request.
  Wait the peer's response
    stage = AUTHENTICATE
    state = PW_SPEKE_WAITA
   }
```

2) speke_authenticate ( ), the authentication process

```
{After received the peer's packet, using eapspeke_decode to decode the peer's
message.
      If state = PW_SPEKE_WAITA
      {
        Compare the proof value to the value in the cache
              if {not match, return 0}
        Set the        state to PW_SPEKE_WAITRESULT to wait the result
        Copy ProofBK to eap message, and send it to the peer
      }
}
```

4.3.5.3 eapclient.c

This program is to simulate the client. PEAP part is omit here, because it's already implemented by many vendors. I added SPEKE part to simulate a client.

```
1) Int main ()
        {Initiate the lib for big number calculation;
        Set the state to PW_SPEKE_WAITB
         Read the req.txt
        Call sendreve_eap ( )
        }

2) respond_eap_speke ( )
        {After received the peer's packets, decode it first, and then define the state
                if state = PW_SPEKE_WAITB
                {compose A and proofAK, and prepare to send to the peer
                }
                else state = PW_SPEKE_RESULT
                {
                 if $Test_{BK}$ = $Proof_{BK}$, then send resp{} to the peer
                 else failed
                }
```

Chapter 5. MEAP testing results

In this chapter, the successful authentication and the failed authentications are described. Ethereal software (www.ethereal.com) is used to capture the packet.

5.1 The setup of running environment

5.1.1 The installation of FreeRADIUS

Download the latest FreeRADIUS version from www.freeRADIUS.org, then unzip to current directory [23].

```
# tar –xzvf freeRADIUS-1.1.2
# ./configure
# make
# make install
```

5.1.2 Change the RADIUS config file to support MEAP

All the configuration files of RADIUS are stored at /usr/local/etc/raddb. Several changes will be made to make the RADIUS work.

1. Clients.conf is the file to set AP's IP address and the secret share between RADIUS server and AP. In our testing, AP is located at the local machine, so the 127.0.0.1 is set.

   Edit clients.conf file.

   ```
   #cd /usr/local/etc/raddb
   #vi clients.conf
   client 127.0.0.1 {
   ```

```
        secret = testing   ← The secret share between RADIUS server and
AP
        shortname = localhost
}
```

2. Users is the file to store the user's name and password. We define a user named "testuser" and password is "testing".

```
Edit users
#cd /usr/local/etc/users
#vi users (Add the user's password as testing)
testuser User-Password == "testing"
```

3. Eap.conf is the file to set which EAP protocol that the RADIUS server should use. EAP modules will use the value of this file to decide which sub modules should be called.

```
Edit eap.conf
#cd /usr/local/etc/users
#vi eap.conf
eap {
    default_eap_type = peap
    peap{
    default_eap_type = speke
    }
}
```

After the above settings, the RADIUS server now ready for MEAP authentication.

5.2 The procedure of the testing of MEAP

The whole procedures of the testing begin with the executed of spekeclient.c, which is located under the folder "freeRADIUS-1.1.2/src/modules/rlm_eap/types/rlm_eap_speke".

1. Start RADIUS server in debugging mode.

   ```
   # radiusd –X
   ```

2. Run spekeclient.c program

   ```
   # ./spekeclient –X 127.0.0.1 auth testing testing < req.txt
   ```

   It calls the sendrecv_eap function and then calls respong_eap_speke to process speke messages.

3. To do the testing, we define a txt file named req.txt to give the initial value of MEAP.

   User-Name = "testuser"

   NAS-IP-Address = 127.0.0.1 ←The address of AP, here I suppose AP and RADIUS on the same computer

   EAP-Code = Response ←EAP type is Response, means client received the server's reply and will begin the authentication

   EAP-Id = 210 ←The first value of EAP message

4. After RADIUS server receives the request from client, it calls the files under /usr/local/raddb to decide the client mode and EAP types. Three files are called, which are eap.conf, users, client.conf.

   In users, the user's password is stored here. RADIUS server will check it with the password that client provided.

   In client.conf, the secret between RADIUS and AP is stored here.

In eap.conf, the EAP module and EAP sub-module is defined here.

When server finds the EAP type is PEAP-SPEKE, it goes to freeRADIUS-1.1.2/src/modules/rlm_eap/types/rlm_eap_speke/ to do the next steps.

5. The main module of EAP needs to know every sub-module under it. Because SPEKE is a new defined sub-module EAP module, SPEKE have to be added in eapcommon.c

6. rlm_eap_speke.c is called to do the main authentication progress, and it calls functions in eap_speke.c to help.

7. If the client provided the right information, the expected result can be print out by the program. But due to the limited time, the client can only support to print out the exchanged messages in SPEKE, the PEAP part is omitted.

5.3 Test 1: The successful authentication

This test shows the successful authentication between the client and server.

Under /src/modules/rlm_eap/types/rlm_eap_speke, run the spekeclient.c with the right parameters. The first **testing** is the secret share between RADIUS and AP, the second **testing** is the client's password. On the mean time, using Ethereal to capture the packets from lo interface.

```
[root@localhost rlm_eap_speke]# ./spekeclient -X 127.0.0.1 auth testing testing
< req.txt
```

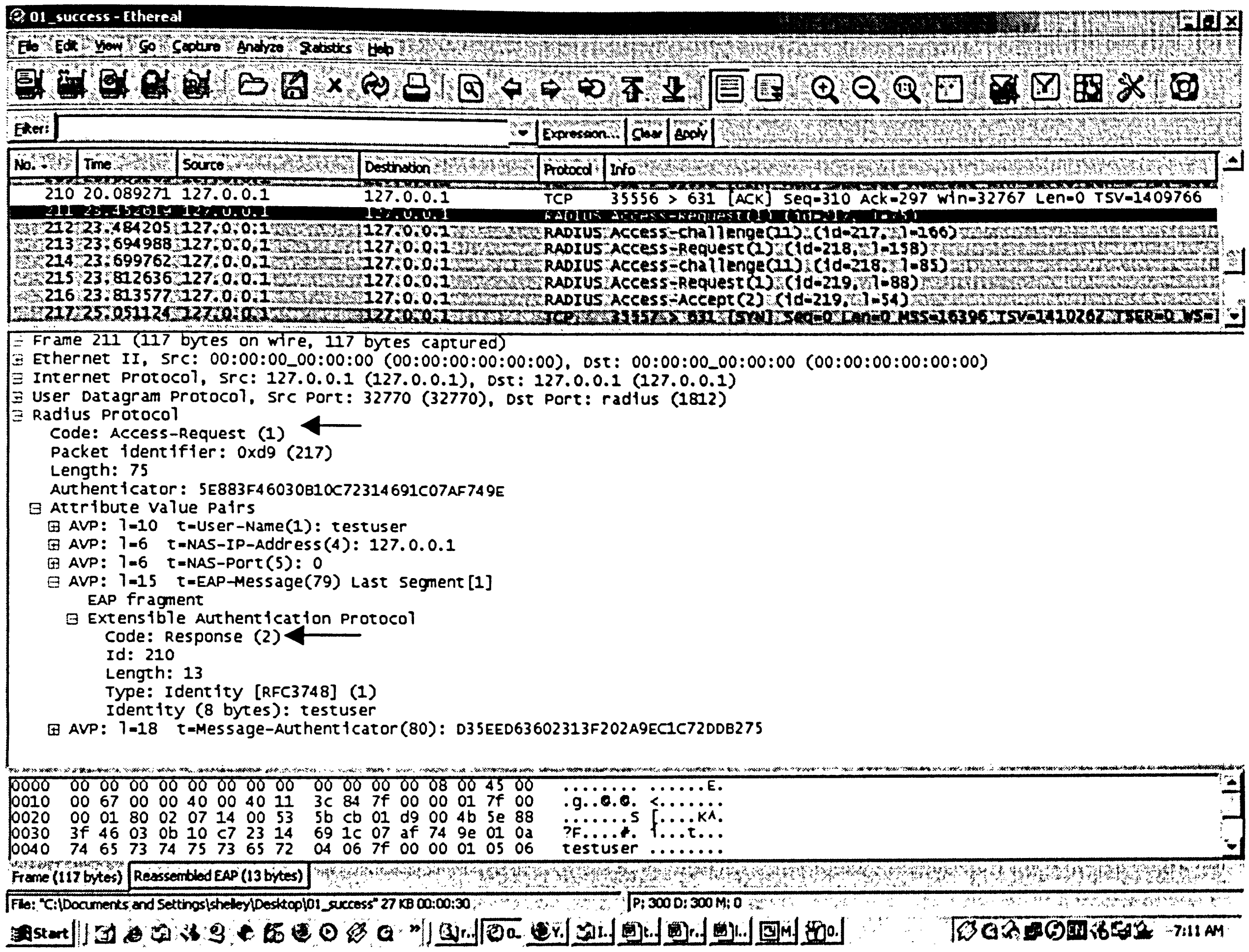
Here let's separate the client side output, and the captured packet by message flow. The client side output shows the EAPOL packet. The Ethereal captured packet shows the RADIUS packet.

1). As indicated by arrow in Figure 5.1, from the Ethereal captured packet 211, we can see that the server received the EAP identity response (code=2, EAP-ID = 210) from client. Then server set the code in RADIUS to 1, mean's it received a request from server.

The output from client side:

```
+++> About to send encoded packet:
          User-Name = "testuser"
          NAS-IP-Address = 127.0.0.1
          EAP-Code = Response
          EAP-Id = 210
          EAP-Type-Identity = "testuser"
          NAS-Port = 0
Received response ID 68, code 11, length = 166
          EAP-Message                                                                    =
0x01d3006c18013431303030303030303030303030303030303030303030303030303030303030303030
303030303030303030303030303031393330002333333230393531383930373131333636383837
38313137363233393830303233383135353331343736353238383530363200
Message-Authenticator = 0x44dec493881e8b44ee6b6913c1814812
          State = 0x4f1adcec177739997719e97bce382421
```

The packet captured by Ethereal is shown as figure 6.1:


01_success - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
Frame (117 bytes) Reassembled EAP (13 bytes)
Start
7:11 AM

```
No.  Time      Source      Destination  Protocol Info
210 20.089271 127.0.0.1   127.0.0.1    TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
211 23.[illegible] 127.0.0.1 127.0.0.1 RADIUS   Access-Request(1) (id=217, l=[illegible])
212 23.484205 127.0.0.1   127.0.0.1    RADIUS   Access-challenge(11) (id=217, l=166)
213 23.694988 127.0.0.1   127.0.0.1    RADIUS   Access-Request(1) (id=218, l=158)
214 23.699762 127.0.0.1   127.0.0.1    RADIUS   Access-challenge(11) (id=218, l=85)
215 23.812636 127.0.0.1   127.0.0.1    RADIUS   Access-Request(1) (id=219, l=88)
216 23.813577 127.0.0.1   127.0.0.1    RADIUS   Access-Accept(2) (id=219, l=54)
217 25.051124 127.0.0.1   127.0.0.1    TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410267 TSER=0 WS=
```

```
Frame 211 (117 bytes on wire, 117 bytes captured)
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
User Datagram Protocol, Src Port: 32770 (32770), Dst Port: radius (1812)
Radius Protocol
  Code: Access-Request (1)
  Packet identifier: 0xd9 (217)
  Length: 75
  Authenticator: 5E883F46030B10C72314691C07AF749E
 Attribute Value Pairs
  AVP: l=10  t=User-Name(1): testuser
  AVP: l=6  t=NAS-IP-Address(4): 127.0.0.1
  AVP: l=6  t=NAS-Port(5): 0
  AVP: l=15  t=EAP-Message(79) Last Segment[1]
    EAP fragment
   Extensible Authentication Protocol
      Code: Response (2)
      Id: 210
      Length: 13
      Type: Identity [RFC3748] (1)
      Identity (8 bytes): testuser
  AVP: l=18  t=Message-Authenticator(80): D35EED63602313F202A9EC1C72DDB275
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 67 00 00 40 00 40 11  3c 84 7f 00 00 01 7f 00   .g..@.@. <.......
0020  00 01 80 02 07 14 00 53  5b cb 01 d9 00 4b 5e 88   .......S [....K^.
0030  3f 46 03 0b 10 c7 23 14  69 1c 07 af 74 9e 01 0a   ?F....#. i...t...
0040  74 65 73 74 75 73 65 72  04 06 7f 00 00 01 05 06   testuser ........
```

```
File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30   P: 300 D: 300 M: 0
```

Figure 5.1 The first packet captured by Ethereal

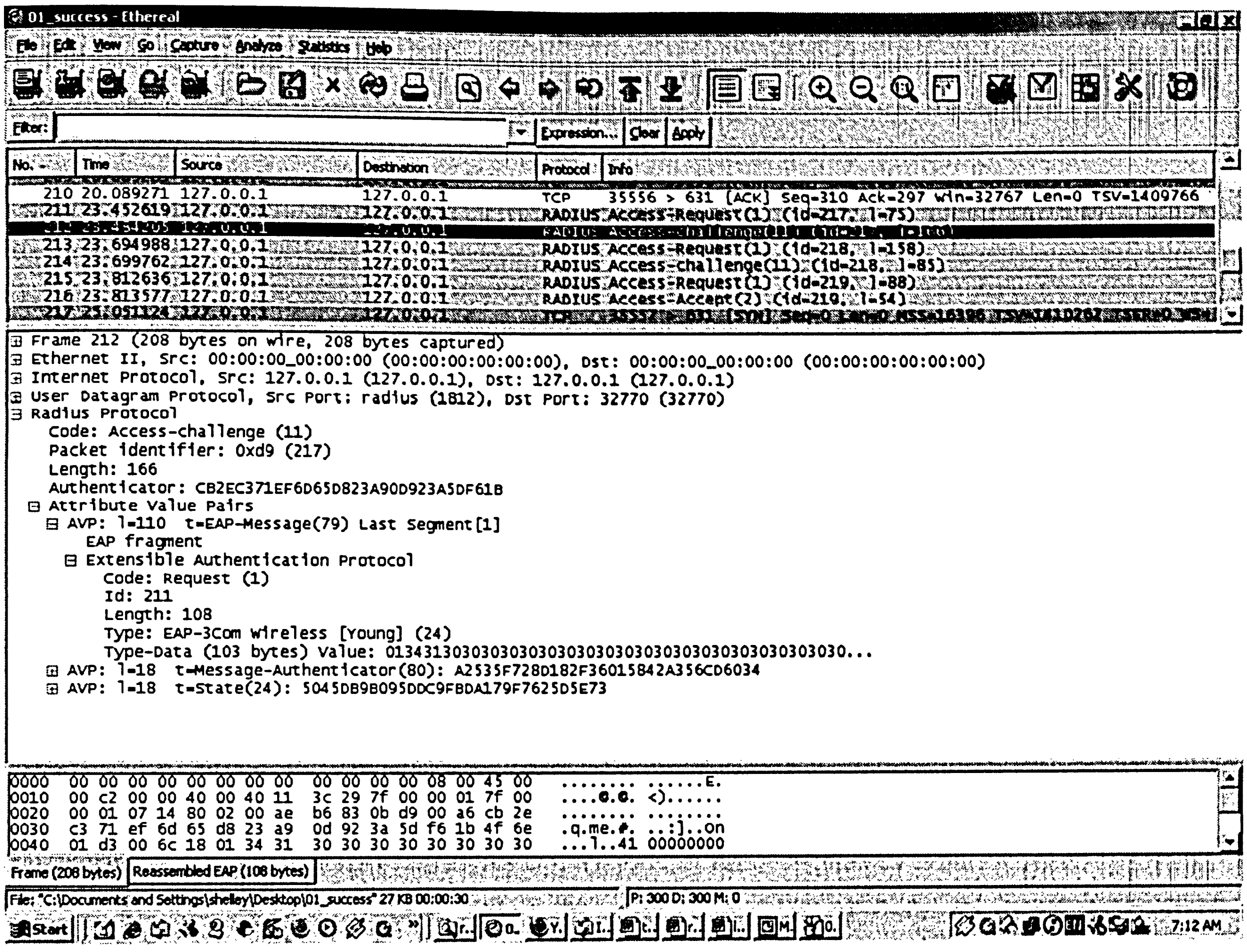
2). Then, the server created the large random number *b*, and calculated *B*. From the Ethereal captured packet 212 that highlighted as indicated, we can see that the server set the packet code to 11. Then sent the challenge type packet (EAP-Id = 211) that encapsulated the {m, B} to client in EAP packet's attribute.

Client received the request, and in EAP type, set the type to Request.

The output from client side:

```
<+++ EAP decoded packet:
            EAP-Message =
0x01d3006c18013431303030303030303030303030303030303030303030303030303030303030
3030303030303030303030303030303030303030313933300023333333637393637303730313 1
363732313231333033353133333938353439353333393931383732393333303638333730
00          Message-Authenticator = 0xa2535f728d182f36015842a356cd6034
            State = 0x5045db9b095ddc9fbda179f7625d5e73
            EAP-Id = 211
            EAP-Code = Request
            Attr-1304 =
0x013431303030303030303030303030303030303030303030303030303030303030303030303030
3030303030303030303030303031393330002333333363739363730373031313637323132
31333033353133333938353439353333393931383732393333303638333730 00
```

The packet captured by Ethereal is shown as figure 5.2:



```
No.  Time      Source     Destination  Protocol Info
  210 20.089271 127.0.0.1  127.0.0.1    TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
  211 23.452619 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=217, l=75)
  212 23.454205 127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=217, l=166)
  213 23.694988 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=218, l=158)
  214 23.699762 127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=218, l=85)
  215 23.812636 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=219, l=88)
  216 23.813577 127.0.0.1  127.0.0.1    RADIUS   Access-Accept(2) (id=219, l=54)
  217 25.051124 127.0.0.1  127.0.0.1    TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410262 TSER=0 WS=0
```

```
⊞ Frame 212 (208 bytes on wire, 208 bytes captured)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
⊞ Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: radius (1812), Dst Port: 32770 (32770)
⊟ Radius Protocol
    Code: Access-challenge (11)
    Packet identifier: 0xd9 (217)
    Length: 166
    Authenticator: CB2EC371EF6D65D823A90D923A5DF61B
  ⊟ Attribute Value Pairs
    ⊟ AVP: l=110  t=EAP-Message(79) Last Segment[1]
        EAP fragment
      ⊟ Extensible Authentication Protocol
          Code: Request (1)
          Id: 211
          Length: 108
          Type: EAP-3Com Wireless [Young] (24)
          Type-Data (103 bytes) Value: 0134313030303030303030303030303030303030303030303030...
    ⊞ AVP: l=18  t=Message-Authenticator(80): A2535F728D182F36015842A356CD6034
    ⊞ AVP: l=18  t=State(24): 5045DB9B095DDC9FBDA179F7625D5E73
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 c2 00 00 40 00 40 11  3c 29 7f 00 00 01 7f 00   ....@.@. <)......
0020  00 01 07 14 80 02 00 ae  b6 83 0b d9 00 a6 cb 2e   ........ ........
0030  c3 71 ef 6d 65 d8 23 a9  0d 92 3a 5d f6 1b 4f 6e   .q.me.#. ..:]..On
0040  01 d3 00 6c 18 01 34 31  30 30 30 30 30 30 30 30   ...l..41 00000000
```

Frame (208 bytes) | Reassembled EAP (108 bytes)

File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30 | P: 300 D: 300 M: 0

01_success - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
Start
7:12 AM

Figure 5.2 The second packet captured by Ethereal

3). On the client side, it created large random number a, and calculated A, K, Proof$_{AK}$, and sent packet (EAP-Id = 211){A, Proof$_{AK}$} to server.

From the Ethereal captured packet 212, we can see that the server received the packet. It set the RADIUS packet type to request.

The output from client

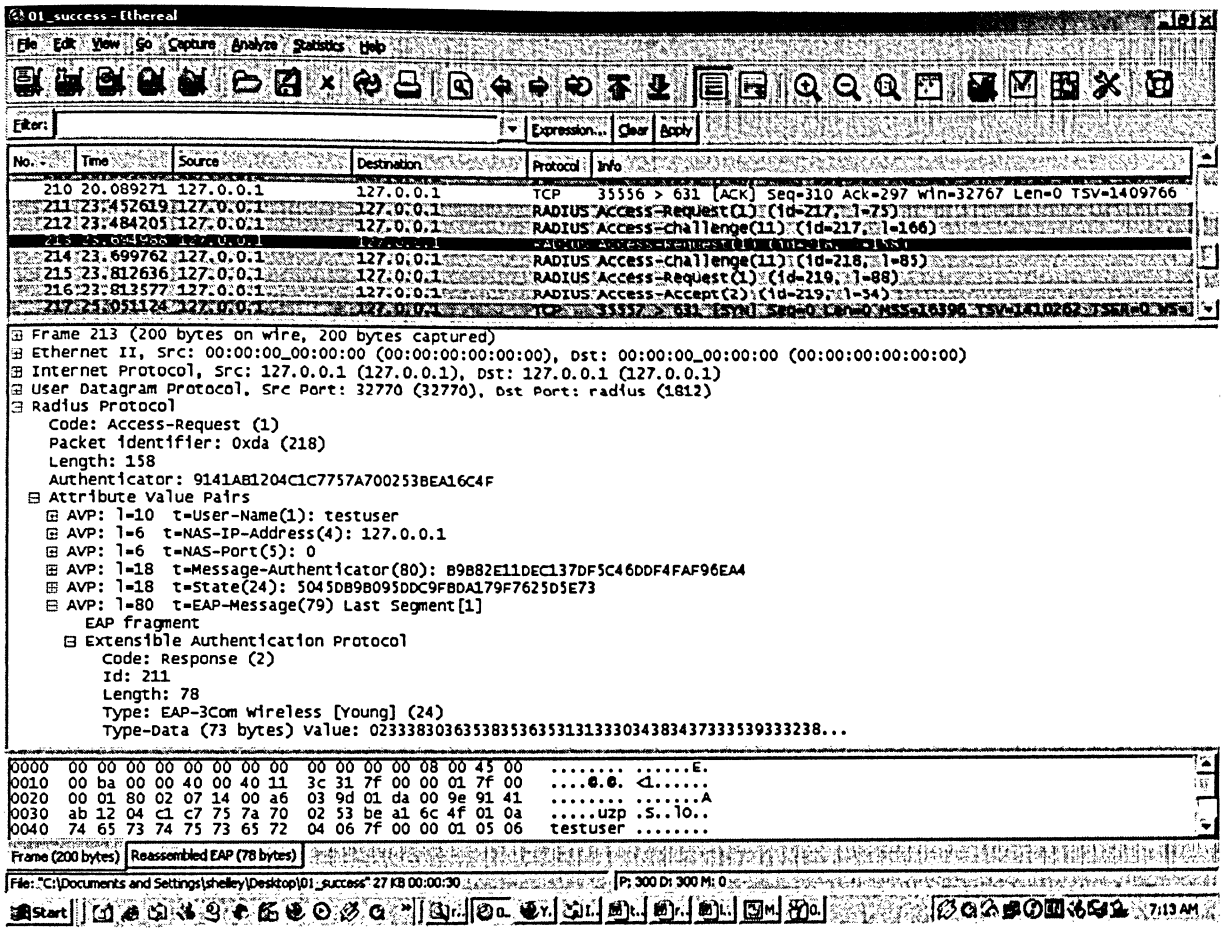
+++> About to send encoded packet:

User-Name = "testuser"

NAS-IP-Address = 127.0.0.1

```
        EAP-Code = Response
        EAP-Id = 211
        NAS-Port = 0
        Message-Authenticator = 0x00000000000000000000000000000000
        Attr-1304 =
0x02333830363538353635313133303438343733353933323838373136313335323332353
03539323737323036373930343735000316a0de1f4e8d7e48f219cbdd04d83090f6342da
572
        State = 0x5045db9b095ddc9fbda179f7625d5e73
Received response ID 218, code 11, length = 85
        EAP-Message =
0x01d4001b18031665f401d17ed89e4640068fe9d8e9edac99359cac
        Message-Authenticator = 0x29a39d87c6d1dee5a661ae86df1def27
        State = 0xc81a43ac5afe1a4767d97d8a118c7507
```

The packet captured by Ethereal is shown as figure 5.3:



01_success - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
Frame (200 bytes)
Reassembled EAP (78 bytes)
Start
7:13 AM

```
No.  Time        Source      Destination   Protocol Info
 210 20.089271 127.0.0.1     127.0.0.1     TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
 211 23.452619 127.0.0.1     127.0.0.1     RADIUS Access-Request(1) (id=217, l=75)
 212 23.484205 127.0.0.1     127.0.0.1     RADIUS Access-challenge(11) (id=217, l=166)
 213 [illegible] 127.0.0.1   127.0.0.1     RADIUS Access-Request[illegible]
 214 23.699762 127.0.0.1     127.0.0.1     RADIUS Access-challenge(11) (id=218, l=85)
 215 23.812636 127.0.0.1     127.0.0.1     RADIUS Access-Request(1) (id=219, l=88)
 216 23.813577 127.0.0.1     127.0.0.1     RADIUS Access-Accept(2) (id=219, l=54)
 217 25.051124 127.0.0.1     127.0.0.1     TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410262 TSER=0 WS=
```

```
⊞ Frame 213 (200 bytes on wire, 200 bytes captured)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
⊞ Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: 32770 (32770), Dst Port: radius (1812)
⊟ Radius Protocol
    Code: Access-Request (1)
    Packet identifier: 0xda (218)
    Length: 158
    Authenticator: 9141AB1204C1C7757A700253BEA16C4F
  ⊟ Attribute Value Pairs
    ⊞ AVP: l=10  t=User-Name(1): testuser
    ⊞ AVP: l=6  t=NAS-IP-Address(4): 127.0.0.1
    ⊞ AVP: l=6  t=NAS-Port(5): 0
    ⊞ AVP: l=18  t=Message-Authenticator(80): B9B82E11DEC137DF5C46DDF4FAF96EA4
    ⊞ AVP: l=18  t=State(24): 5045DB9B095DDC9FBDA179F7625D5E73
    ⊟ AVP: l=80  t=EAP-Message(79) Last Segment[1]
        EAP fragment
      ⊟ Extensible Authentication Protocol
          Code: Response (2)
          Id: 211
          Length: 78
          Type: EAP-3Com Wireless [Young] (24)
          Type-Data (73 bytes) Value: 023338303635383536353131333034383437333539333238...
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 ba 00 00 40 00 40 11  3c 31 7f 00 00 01 7f 00   ....@.@. <1......
0020  00 01 80 02 07 14 00 a6  03 9d 01 da 00 9e 91 41   ........ .......A
0030  ab 12 04 c1 c7 75 7a 70  02 53 be a1 6c 4f 01 0a   .....uzp .S..lO..
0040  74 65 73 74 75 73 65 72  04 06 7f 00 00 01 05 06   testuser ........
```

```
File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30    P: 300 D: 300 M: 0
```

Figure 5.3 The third packet captured by Ethereal

4). From the Ethereal captured packet 214, we can see that on server side, it created K, $Test_{AK}$, and compared the value of $Test_{AK}$ and the received $Proof_{AK}$. Because the value is equal, it sent packet (EAP-Id = 211){$Proof_{BK}$} to client.

Client received the packet, and then changed the type to request.

The output from client:

<+++ EAP decoded packet:

EAP-Message =

0x01d4001b18031665f401d17ed89e4640068fe9d8e9edac99359cac

Message-Authenticator = 0x29a39d87c6d1dee5a661ae86df1def27

State = 0xc81a43ac5afe1a4767d97d8a118c7507

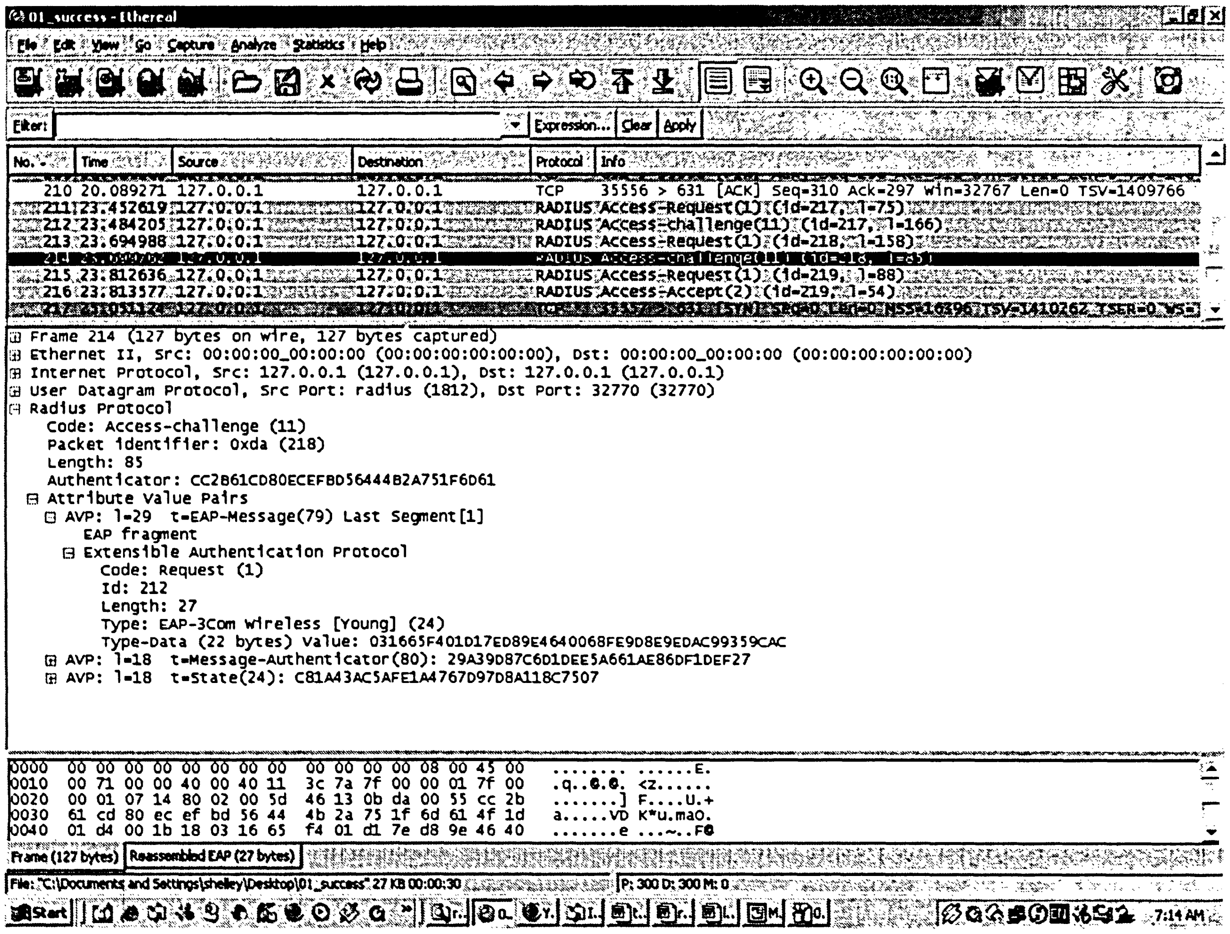
EAP-Id = 212

EAP-Code = Request

Attr-1304 = 0x031665f401d17ed89e4640068fe9d8e9edac99359cac

speke success

The packet captured by Ethereal is shown as figure 5.4:



```
No.  Time      Source     Destination  Protocol Info
 210 20.089271 127.0.0.1  127.0.0.1    TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
 211 23.452619 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=217, l=75)
 212 23.484205 127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=217, l=166)
 213 23.694988 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=218, l=158)
 214 23.699702 127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=218, l=85)
 215 23.812636 127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=219, l=88)
 216 23.813577 127.0.0.1  127.0.0.1    RADIUS   Access-Accept(2) (id=219, l=54)
 217 25.051124 127.0.0.1  127.0.0.1    TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410262 TSER=0 WS=
```

```
⊞ Frame 214 (127 bytes on wire, 127 bytes captured)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
⊞ Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: radius (1812), Dst Port: 32770 (32770)
⊟ Radius Protocol
    Code: Access-challenge (11)
    Packet identifier: 0xda (218)
    Length: 85
    Authenticator: CC2B61CD80ECEFBD56444B2A751F6D61
  ⊟ Attribute Value Pairs
    ⊟ AVP: l=29  t=EAP-Message(79) Last Segment[1]
        EAP fragment
      ⊟ Extensible Authentication Protocol
          Code: Request (1)
          Id: 212
          Length: 27
          Type: EAP-3Com Wireless [Young] (24)
          Type-Data (22 bytes) Value: 031665F401D17ED89E4640068FE9D8E9EDAC99359CAC
    ⊞ AVP: l=18  t=Message-Authenticator(80): 29A39D87C6D1DEE5A661AE86DF1DEF27
    ⊞ AVP: l=18  t=State(24): C81A43AC5AFE1A4767D97D8A118C7507
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 71 00 00 40 00 40 11  3c 7a 7f 00 00 01 7f 00   .q..@.@. <z......
0020  00 01 07 14 80 02 00 5d  46 13 0b da 00 55 cc 2b   .......] F....U.+
0030  61 cd 80 ec ef bd 56 44  4b 2a 75 1f 6d 61 4f 1d   a.....VD K*u.maO.
0040  01 d4 00 1b 18 03 16 65  f4 01 d1 7e d8 9e 46 40   .......e ...~..F@
```

Frame (127 bytes) | Reassembled EAP (27 bytes)

File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30 | P: 300 D: 300 M: 0

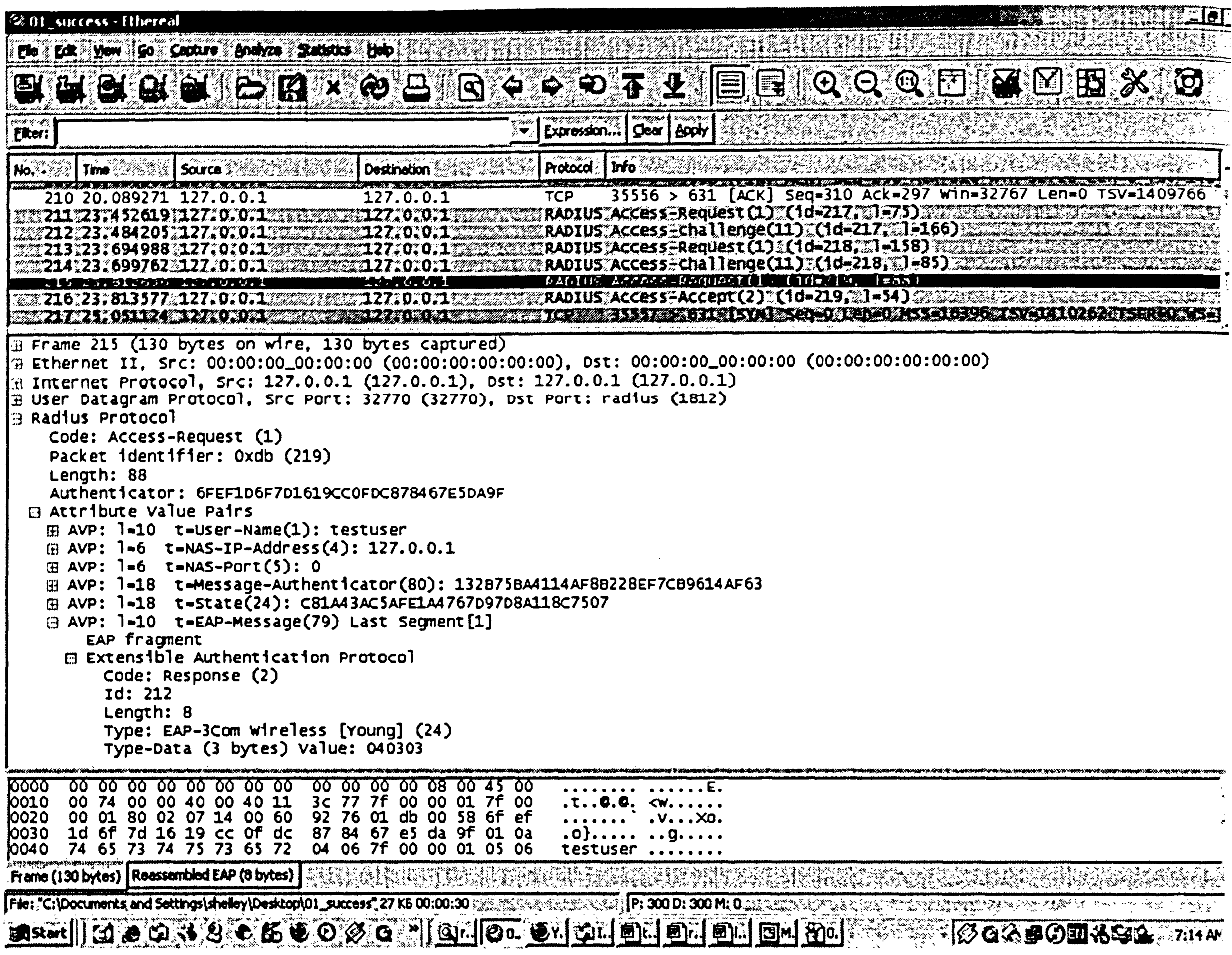
Figure 5.4 The fourth packet captured by Ethereal

5). Client sent an empty response message to the server. From the Ethereal captured packet 215, we can see that the server received the packet, and then set the packet type to response.

The output from the client:

```
+++> About to send encoded packet:
        User-Name = "testuser"
        NAS-IP-Address = 127.0.0.1
        EAP-Code = Response
        EAP-Id = 212
        NAS-Port = 0
        Message-Authenticator = 0x00000000000000000000000000000000
        State = 0xc81a43ac5afe1a4767d97d8a118c7507
        Attr-1304 = 0x040303
Received response ID 219, code 2, length = 54
        EAP-Message = 0x03d40004
        Message-Authenticator = 0x54afe11fcb0f5fd309c004c7b3b53c44
        User-Name = "testuser"
```

The packet captured by Ethereal is shown as figure 5.5:


01_success - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
Frame (130 bytes) Reassembled EAP (8 bytes)
Start
7:14 AM

```
No. Time      Source     Destination Protocol Info
210 20.089271 127.0.0.1  127.0.0.1   TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
211 23.452619 127.0.0.1  127.0.0.1   RADIUS   Access-Request(1) (id=217, l=75)
212 23.484205 127.0.0.1  127.0.0.1   RADIUS   Access-challenge(11) (id=217, l=166)
213 23.694988 127.0.0.1  127.0.0.1   RADIUS   Access-Request(1) (id=218, l=158)
214 23.699762 127.0.0.1  127.0.0.1   RADIUS   Access-Challenge(11) (id=218, l=85)
215 23.[illegible] 127.0.0.1 127.0.0.1 RADIUS Access-Request(1) (id=219, l=88)
216 23.813577 127.0.0.1  127.0.0.1   RADIUS   Access-Accept(2) (id=219, l=54)
217 25.051124 127.0.0.1  127.0.0.1   TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410262 TSER=0 WS=

Frame 215 (130 bytes on wire, 130 bytes captured)
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
User Datagram Protocol, Src Port: 32770 (32770), Dst Port: radius (1812)
Radius Protocol
  Code: Access-Request (1)
  Packet identifier: 0xdb (219)
  Length: 88
  Authenticator: 6FEF1D6F7D1619CC0FDC878467E5DA9F
 Attribute Value Pairs
   AVP: l=10  t=User-Name(1): testuser
   AVP: l=6  t=NAS-IP-Address(4): 127.0.0.1
   AVP: l=6  t=NAS-Port(5): 0
   AVP: l=18  t=Message-Authenticator(80): 132B75BA4114AF8B228EF7CB9614AF63
   AVP: l=18  t=State(24): C81A43AC5AFE1A4767D97D8A118C7507
   AVP: l=10  t=EAP-Message(79) Last Segment[1]
     EAP fragment
    Extensible Authentication Protocol
      Code: Response (2)
      Id: 212
      Length: 8
      Type: EAP-3Com Wireless [Young] (24)
      Type-Data (3 bytes) Value: 040303

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 74 00 00 40 00 40 11  3c 77 7f 00 00 01 7f 00   .t..@.@. <w......
0020  00 01 80 02 07 14 00 60  92 76 01 db 00 58 6f ef   .......` .v...Xo.
0030  1d 6f 7d 16 19 cc 0f dc  87 84 67 e5 da 9f 01 0a   .o}..... ..g.....
0040  74 65 73 74 75 73 65 72  04 06 7f 00 00 01 05 06   testuser ........

File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30   P: 300 D: 300 M: 0
```

Figure 5.5 The fifth packet captured by Ethereal

6). After receive the empty EAP request packet, the server sent the success message to client to indicate the successfully authentication. We can see it from the Ethereal captured packet 216.

The client received the success packet. Then this is the end of the successful authentication

The output from client

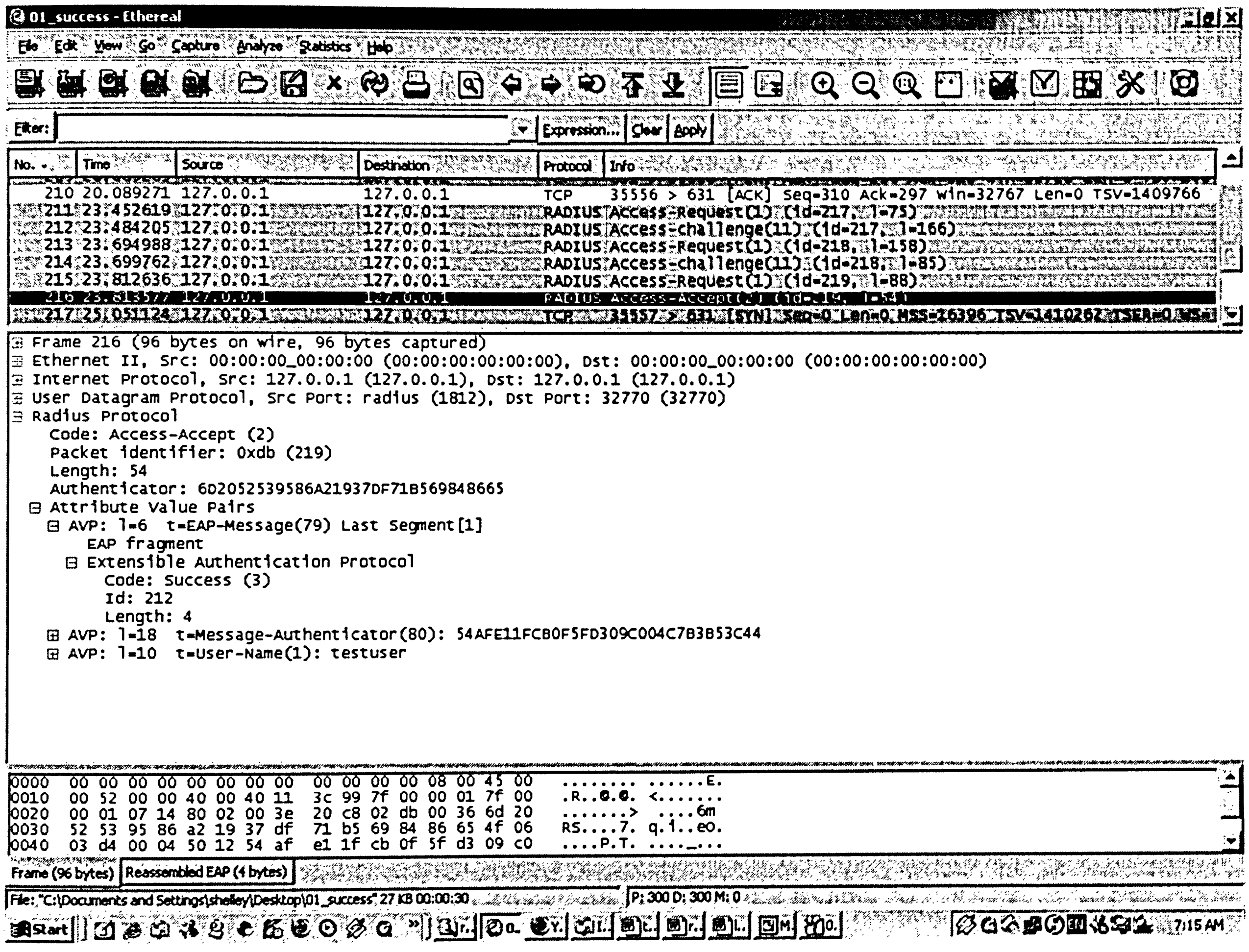
```
EAP-Message = 0x03d40004
Message-Authenticator = 0x54afe11fcb0f5fd309c004c7b3b53c44
```

User-Name = "testuser"

EAP-Id = 212

EAP-Code = Success

The packet captured by Ethereal is shown as figure 5.6:



```
No.  Time       Source     Destination  Protocol Info
210  20.089271  127.0.0.1  127.0.0.1    TCP      35556 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=1409766
211  23.452619  127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=217, l=75)
212  23.484205  127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=217, l=166)
213  23.694988  127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=218, l=158)
214  23.699762  127.0.0.1  127.0.0.1    RADIUS   Access-challenge(11) (id=218, l=85)
215  23.812636  127.0.0.1  127.0.0.1    RADIUS   Access-Request(1) (id=219, l=88)
216  23.813577  127.0.0.1  127.0.0.1    RADIUS   Access-Accept(2) (id=219, l=54)
217  25.051124  127.0.0.1  127.0.0.1    TCP      35557 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1410262 TSER=0 WS=

Frame 216 (96 bytes on wire, 96 bytes captured)
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
User Datagram Protocol, Src Port: radius (1812), Dst Port: 32770 (32770)
Radius Protocol
   Code: Access-Accept (2)
   Packet identifier: 0xdb (219)
   Length: 54
   Authenticator: 6D2052539586A21937DF71B569848665
 Attribute Value Pairs
   AVP: l=6  t=EAP-Message(79) Last Segment[1]
     EAP fragment
     Extensible Authentication Protocol
       Code: Success (3)
       Id: 212
       Length: 4
   AVP: l=18  t=Message-Authenticator(80): 54AFE11FCB0F5FD309C004C7B3B53C44
   AVP: l=10  t=User-Name(1): testuser

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 52 00 00 40 00 40 11  3c 99 7f 00 00 01 7f 00   .R..@.@. <.......
0020  00 01 07 14 80 02 00 3e  20 c8 02 db 00 36 6d 20   .......>  ....6m
0030  52 53 95 86 a2 19 37 df  71 b5 69 84 86 65 4f 06   RS....7. q.i..eO.
0040  03 d4 00 04 50 12 54 af  e1 1f cb 0f 5f d3 09 c0   ....P.T. ...._...

Frame (96 bytes) | Reassembled EAP (4 bytes)
File: "C:\Documents and Settings\shelley\Desktop\01_success" 27 KB 00:00:30   P: 300 D: 300 M: 0
```

01_success - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
Start
7:15 AM

Figure 5.6 The last packet captured by Ethereal

5.3 Test 2: The failed authentication-wrong shared secret

Under /src/modules/rlm_eap/types/rlm_eap_speke, we run the spekeclient with one wrong parameter. The first testing is the correct secret share between RADIUS and AP. This time I changed it to "test", to see the procedures of authentication.

According to the design of testing, it simulates the procedures after shared secret matched. That means, if the shared secret is not match, the authentication will not begin. Figure 5.7 shows the flow when RADIUS server receive the wrong shared secret

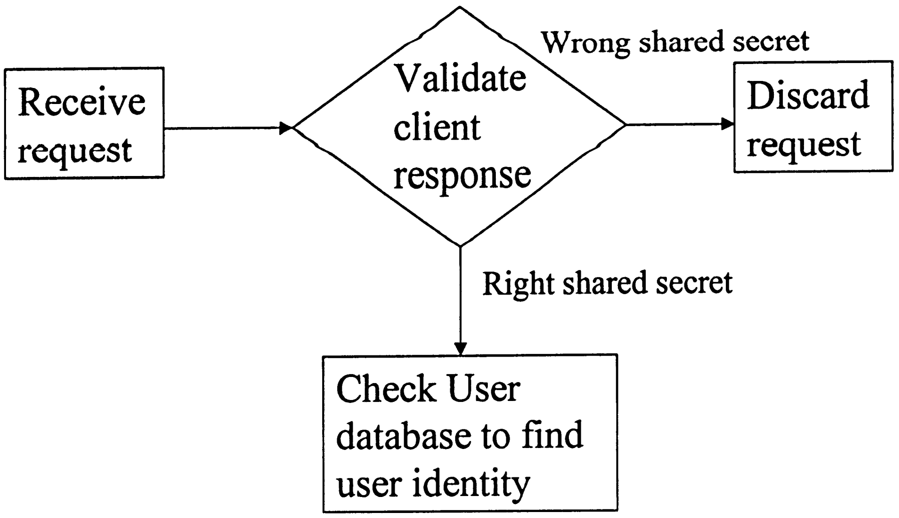


Figure 5.7 Wrong shared secret flow

Now from the client output, we can see it doesn't received server's response.

```
[root@localhost rlm_eap_speke]# ./spekeclient -X 127.0.0.1 auth test testing
< req.txt
```

The output from client shows it didn't received packet from server.

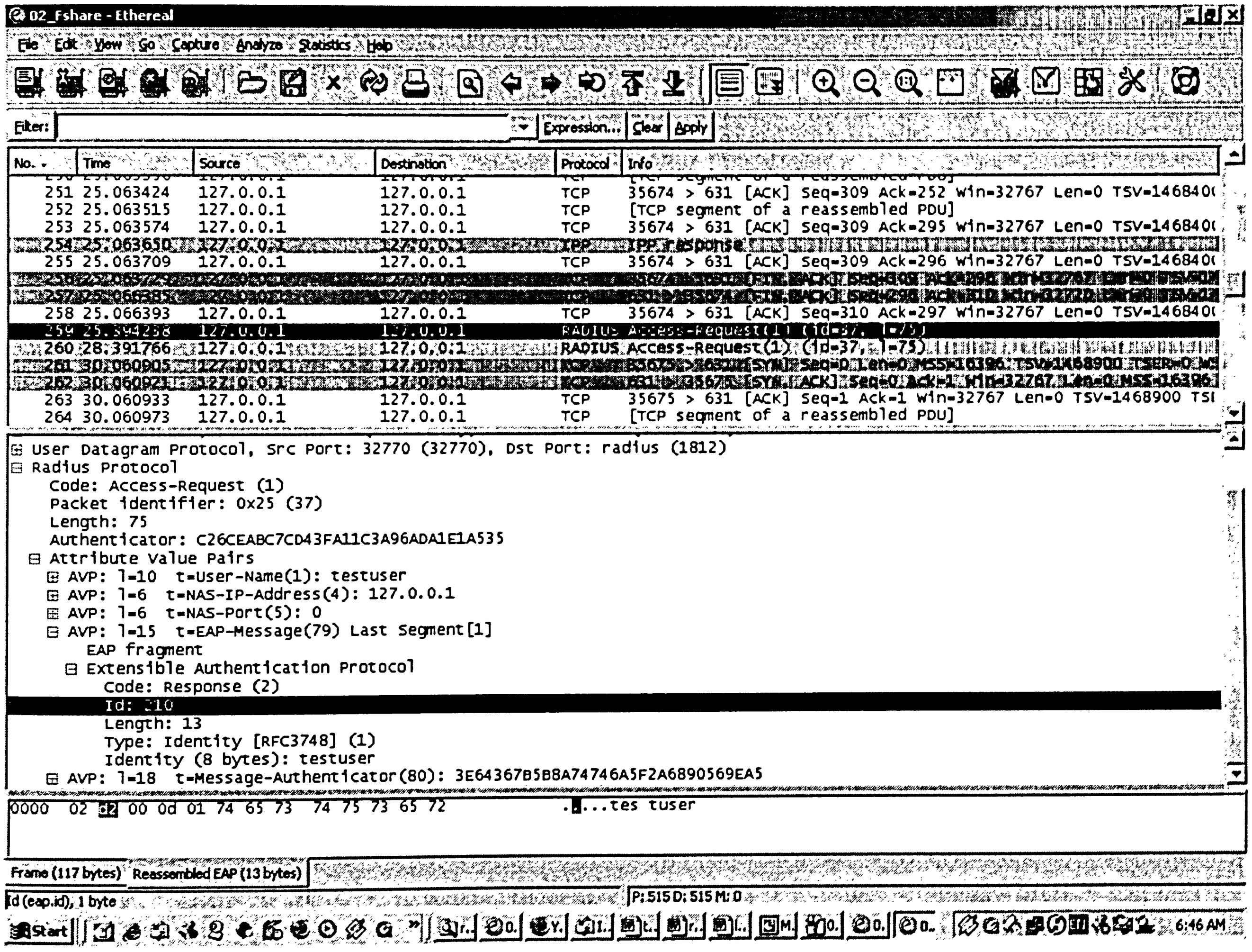
```
+++> About to send encoded packet:
        User-Name = "testuser"
        NAS-IP-Address = 127.0.0.1
        EAP-Code = Response
        EAP-Id = 210
        EAP-Type-Identity = "testuser"
```

NAS-Port = 0

radclient: no response from server

Form the Ethereal packet, we can only see one (the other one is same) request from client, and then because it is a wrong shared secret, RADIUS server drop the packet.

The packet captured by Ethereal is shown as figure 5.8:


02_Fshare - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. -  Time       Source      Destination  Protocol Info
  251 25.063424  127.0.0.1   127.0.0.1    TCP      35674 > 631 [ACK] Seq=309 Ack=252 Win=32767 Len=0 TSV=146840(
  252 25.063515  127.0.0.1   127.0.0.1    TCP      [TCP segment of a reassembled PDU]
  253 25.063574  127.0.0.1   127.0.0.1    TCP      35674 > 631 [ACK] Seq=309 Ack=295 Win=32767 Len=0 TSV=146840(
  254 25.063650  127.0.0.1   127.0.0.1    IPP      IPP response
  255 25.063709  127.0.0.1   127.0.0.1    TCP      35674 > 631 [ACK] Seq=309 Ack=296 Win=32767 Len=0 TSV=146840(
  256 25.063729  [illegible]                        35674 > 631 [FIN, ACK] Seq=309 Ack=296 Win=32767 [illegible]
  257 25.066385  [illegible]                        631 > 35674 [FIN, ACK] Seq=296 Ack=310 Win=32767 [illegible]
  258 25.066393  127.0.0.1   127.0.0.1    TCP      35674 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=146840(
  259 25.394238  127.0.0.1   127.0.0.1    RADIUS   Access-Request(1) (id=37, l=75)
  260 28.391766  127.0.0.1   127.0.0.1    RADIUS   Access-Request(1) (id=37, l=75)
  261 30.060905  127.0.0.1   127.0.0.1    TCP      35675 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1468900 TSER=0 WS
  262 30.060921  127.0.0.1   127.0.0.1    TCP      631 > 35675 [SYN, ACK] Seq=0 Ack=1 Win=32767 Len=0 MSS=16396
  263 30.060933  127.0.0.1   127.0.0.1    TCP      35675 > 631 [ACK] Seq=1 Ack=1 Win=32767 Len=0 TSV=1468900 TSI
  264 30.060973  127.0.0.1   127.0.0.1    TCP      [TCP segment of a reassembled PDU]
```

```
⊞ User Datagram Protocol, Src Port: 32770 (32770), Dst Port: radius (1812)
⊟ Radius Protocol
    Code: Access-Request (1)
    Packet identifier: 0x25 (37)
    Length: 75
    Authenticator: C26CEABC7CD43FA11C3A96ADA1E1A535
  ⊟ Attribute Value Pairs
    ⊞ AVP: l=10  t=User-Name(1): testuser
    ⊞ AVP: l=6  t=NAS-IP-Address(4): 127.0.0.1
    ⊞ AVP: l=6  t=NAS-Port(5): 0
    ⊟ AVP: l=15  t=EAP-Message(79) Last Segment[1]
        EAP fragment
      ⊟ Extensible Authentication Protocol
          Code: Response (2)
          Id: 210
          Length: 13
          Type: Identity [RFC3748] (1)
          Identity (8 bytes): testuser
    ⊞ AVP: l=18  t=Message-Authenticator(80): 3E64367B5B8A74746A5F2A6890569EA5
```

```
0000  02 d2 00 0d 01 74 65 73  74 75 73 65 72          .....tes tuser
```

Figure 5.8 The packets captured by Ethereal when shared secret is not match

5.4 Test 3: The failed authentication: wrong user's password

Under /src/modules/rlm_eap/types/rlm_eap_speke, we run the spekeclient.c with one wrong parameter. The second testing is the correct user's password. This time we changed it to "test", to see the procedures of authentication.

Figure 5.9 shows the flow when RADIUS server receive the wrong password.

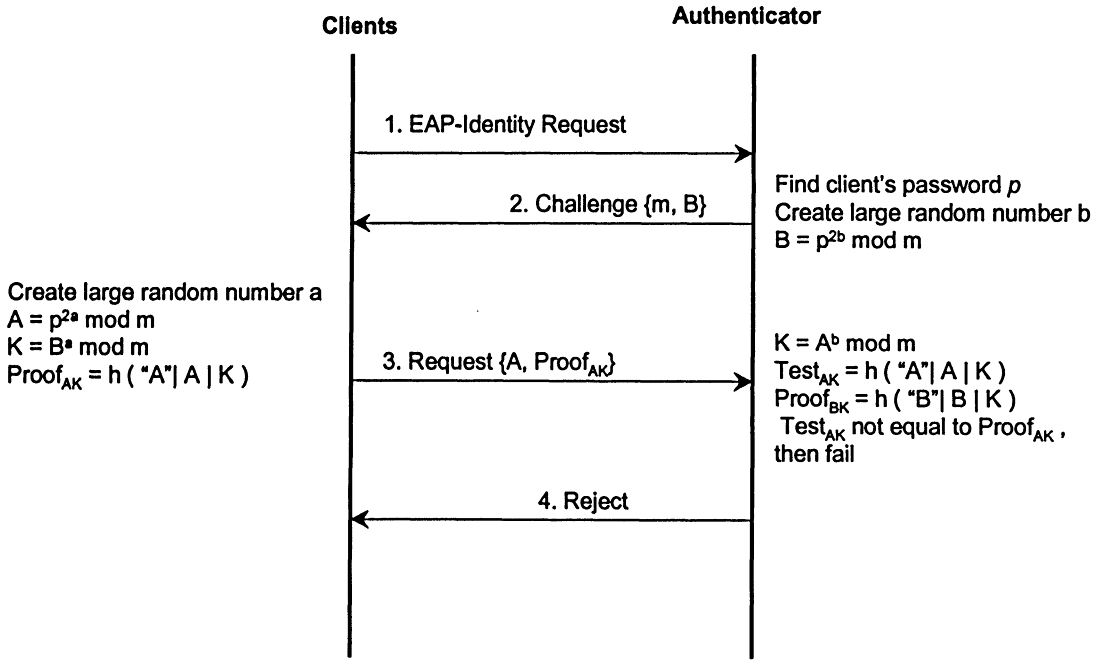

Clients
Authenticator
1. EAP-Identity Request
2. Challenge {m, B}
Find client's password p
Create large random number b
$B = p^{2b} \bmod m$
Create large random number a
$A = p^{2a} \bmod m$
$K = B^a \bmod m$
$Proof_{AK} = h(\text{"A"} | A | K)$
3. Request {A, $Proof_{AK}$}
$K = A^b \bmod m$
$Test_{AK} = h(\text{"A"} | A | K)$
$Proof_{BK} = h(\text{"B"} | B | K)$
$Test_{AK}$ not equal to $Proof_{AK}$, then fail
4. Reject

Figure 5.9 Message flow when RADIUS received a wrong password

The whole procedure is described as following:

1) Client send packet (EAP-ID = 210) to server, in RADIUS request type packet.
2) Server found the right password p from the database and send {m, B} in Challenge type packet to client (EAP-ID = 211).

3) Client received this packet and create the large random number a, and calculate A by a and it's own password (the wrong one), then send response to server (EAP-ID = 211). In RADIUS packet, it's a request type massage.
4) On server side, because Test$_{AK}$ not equal to Proof$_{AK}$, the authentication failed. It sent the packet (EAP-ID = 211) to client to show authentication failed.

The output from client side:

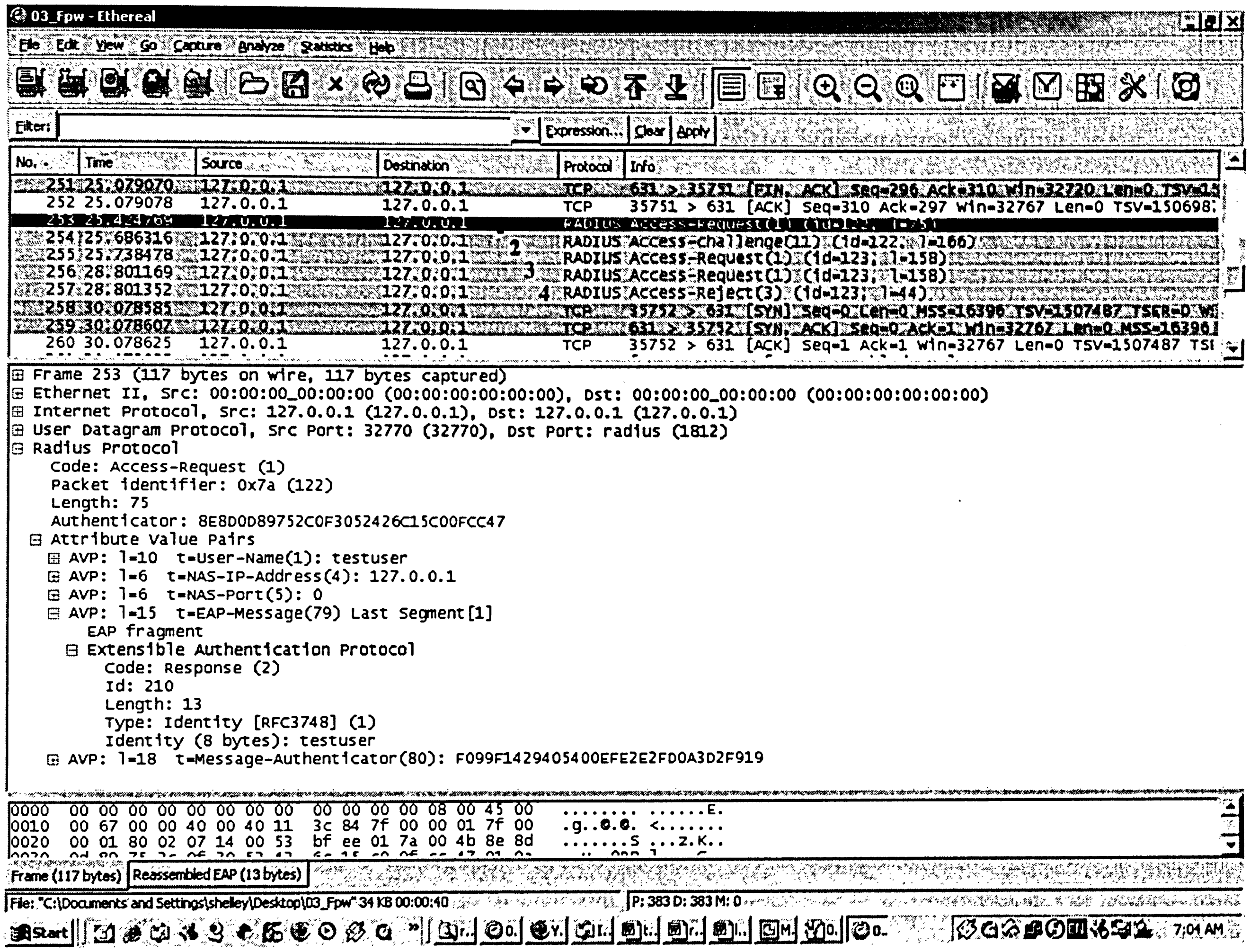
```
+++> About to send encoded packet:              →1
          User-Name = "testuser"
          NAS-IP-Address = 127.0.0.1
          EAP-Code = Response
          EAP-Id = 210
          EAP-Type-Identity = "testuser"
          NAS-Port = 0
Received response ID 122, code 11, length = 166
          EAP-Message =
0x01d3006c180134313030303030303030303030303030303030303030303030303030303030
303030303030303030303030303030303030313933000233323839333831383631353935
3534383932303835323635373338343631353233373638343234323534353032323231 36
00          Message-Authenticator = 0xb5ddfb6fe9fcda31a356db620da3f0e5
          State = 0xa1bf1630c1a6b9d028d666d68576dd61
<+++ EAP decoded packet:              →2
          EAP-Message =
0x01d3006c180134313030303030303030303030303030303030303030303030303030303030
303030303030303030303030303030303030313933000233323839333831383631353935
353438393230383532363537333834363135323337363834323432353435303232323136
00          Message-Authenticator = 0xb5ddfb6fe9fcda31a356db620da3f0e5
```

```
            State = 0xa1bf1630c1a6b9d028d666d68576dd61
            EAP-Id = 211
            EAP-Code = Request
            Attr-1304 =
0x0134313030303030303030303030303030303030303030303030303030303030303030
303030303030303030303030303139330002333238393338313836313539353534383932
3038353236353733383436313532333736383432343235343530323232313600

+++> About to send encoded packet:      →3
            User-Name = "testuser"
            NAS-IP-Address = 127.0.0.1
            EAP-Code = Response
            EAP-Id = 211
            NAS-Port = 0
            Message-Authenticator = 0x00000000000000000000000000000000
            Attr-1304 =
0x0233363836373639373930323538383334343331393834363430313633323631343433
3034333739383232353138393538300003162 34ba65aa3d0bc50adbc883d1bca7c18c207d
98b
            State = 0xa1bf1630c1a6b9d028d666d68576dd61
Received response ID 123, code 3, length = 44
            EAP-Message = 0x04d30004
            Message-Authenticator = 0x6837bb3bb5222223f851f7e4643b0c2f
<+++ EAP decoded packet:    →4
            EAP-Message = 0x04d30004
            Message-Authenticator = 0x6837bb3bb5222223f851f7e4643b0c2f
            EAP-Id = 211
```

EAP-Code = Failure

The packet captured by Ethereal is shown as figure 5.10:


03_Fpw - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply
Frame (117 bytes)
Reassembled EAP (13 bytes)
Start
7:04 AM

```
No.  Time        Source      Destination  Protocol  Info
251 25.079070  127.0.0.1   127.0.0.1    TCP     631 > 35751 [FIN, ACK] Seq=296 Ack=310 Win=32720 Len=0 TSV=15
252 25.079078  127.0.0.1   127.0.0.1    TCP     35751 > 631 [ACK] Seq=310 Ack=297 Win=32767 Len=0 TSV=150698:
253 25.423769  127.0.0.1   127.0.0.1    RADIUS  Access-Request(1) (id=122, l=75)
254 25.686316  127.0.0.1   127.0.0.1  2 RADIUS  Access-challenge(11) (id=122, l=166)
255 25.738478  127.0.0.1   127.0.0.1    RADIUS  Access-Request(1) (id=123, l=158)
256 28.801169  127.0.0.1   127.0.0.1  3 RADIUS  Access-Request(1) (id=123, l=158)
257 28.801352  127.0.0.1   127.0.0.1  4 RADIUS  Access-Reject(3) (id=123, l=44)
258 30.078585  127.0.0.1   127.0.0.1    TCP     35752 > 631 [SYN] Seq=0 Len=0 MSS=16396 TSV=1507487 TSER=0 W
259 30.078607  127.0.0.1   127.0.0.1    TCP     631 > 35752 [SYN, ACK] Seq=0 Ack=1 Win=32767 Len=0 MSS=16396
260 30.078625  127.0.0.1   127.0.0.1    TCP     35752 > 631 [ACK] Seq=1 Ack=1 Win=32767 Len=0 TSV=1507487 TSI
```

```
⊞ Frame 253 (117 bytes on wire, 117 bytes captured)
⊞ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
⊞ Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
⊞ User Datagram Protocol, Src Port: 32770 (32770), Dst Port: radius (1812)
⊟ Radius Protocol
    Code: Access-Request (1)
    Packet identifier: 0x7a (122)
    Length: 75
    Authenticator: 8E8D0D89752C0F3052426C15C00FCC47
  ⊟ Attribute Value Pairs
    ⊞ AVP: l=10  t=User-Name(1): testuser
    ⊞ AVP: l=6  t=NAS-IP-Address(4): 127.0.0.1
    ⊞ AVP: l=6  t=NAS-Port(5): 0
    ⊟ AVP: l=15  t=EAP-Message(79) Last Segment[1]
        EAP fragment
      ⊟ Extensible Authentication Protocol
          Code: Response (2)
          Id: 210
          Length: 13
          Type: Identity [RFC3748] (1)
          Identity (8 bytes): testuser
    ⊞ AVP: l=18  t=Message-Authenticator(80): F099F1429405400EFE2E2FD0A3D2F919
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 67 00 00 40 00 40 11  3c 84 7f 00 00 01 7f 00   .g..@.@. <.......
0020  00 01 80 02 07 14 00 53  bf ee 01 7a 00 4b 8e 8d   .......S ...z.K..
```

```
File: "C:\Documents and Settings\shelley\Desktop\03_Fpw" 34 KB 00:00:40    P: 383 D: 383 M: 0
```

Figure 5.10 The packets captured by Ethereal when password is not match

Chapter 6. Conclusion and Future work

This thesis compares the popular authentication protocols in WLAN. Selection of an authentication method is the key decision in securing a wireless LAN deployment. The authentication method drives the choice of authentication server, which in turn drives the choice of client software.

MEAP combines PEAP and SPEKE to provide convenient authentication over different platforms. The common PEAP combines with EAP-MSCHAPv2 that limited PEAP to windows based operation system. MEAP maintains the simplicity with the required level of security. The user is authenticated to the network using ordinary username and password credentials, which are secure against an interception by enclosing them in the TLS security wrapper. MEAP requires only server-side certificates, uses TLS for the secure tunnel, and extends the EAP-TLS beyond the finished message exchange to add SPEKE authentication mechanism.

For the security consideration, MEAP are able to against MITM and Spoof attacks.

- To against MITM: A man-in-the-middle [15] attack can spoof the client to authenticate to it instead of the real EAP server, and forward the authentication to the real server over a protected tunnel. Since the attacker has access to the keys derived from the tunnel, it can gain access to the network. MEAP can prevent this attack by using the keys generated by the inner EAP method in the crypto-binding exchange. Since all sequence negotiations and exchanges are protected by TLS channel, MEAP is immune to snooping and MITM attacks with the use of

Crypto-Binding TLV. If the Crypto-Binding TLV failed validation, the authentication failed, and stopped the subsequent procedures.

- To against spoofing: As we introduced in Chapter1, MAC address spoofing is another kind of attacking in WLAN. The MAC address is a hardware address that uniquely identifies each node of a network. An attacker wishing to disrupt a wireless network can present himself as an authorized client by using an altered MAC address. As nearly all wireless Network Interface Cards (NIC) permit changing their MAC address to an arbitrary value through vendor-supplied drivers, open-source drivers or various application programming frameworks. An attacker using these tools can launch denial of service attacks, bypass access control mechanisms, or falsely advertise services to wireless clients. MEAP requires the use of user authentication like user name and password before accessing the network. So assume the attacker could associate with the AP as it has a valid user's MAC address, thus it pass the first line of defense, MAC address filtering. The attacker still was prompted for the user credentials. This stage could not be by-passed and the attacker could not access the network. If the attacker only know user's MAC address, he cannot gain access to the WLAN that make use of MEAP.

The following table gives a general comparison of EAP methods that are described in this thesis.

	EAP-TLS	**EAP-TTLS**	**PEAP**	**SPEKE**	**MEAP**
Supported client platforms	Linux, Mac OS, Windows 95/98/ME, NT/2000/XP	Linux, Mac OS, Windows 95/98/ME, NT/2000/XP	Windows XP	None	Linux, Mac OS, Windows 95/98/ME, NT/2000/XP
Basic protocol structure	Establish TLS session and validate certificates on both client and server	Two phase: 1. Establish TLS between slient and TTLS server 2. Exchange attribute-value pairs between client and server	Two phase: 1. Establish TLS between client and PEAP server 2. Run MSCHAP exchange over TLS tunnel	Rely on exponentiation involving large random numbers modulo a large prime number	Two phase: 1. Establish TLS between client and PEAP server 2. Run SPEKE exchange over TLS tunnel
Protection of user identity exchange	No	Yes; protected by TLS	Yes; protected by TLS	No	Yes; protected by TLS
Certification Authority	Both on server/client	Only server side	Only server side	none	Only server side
Flexible level to install	Most difficult	Medium	Medium	Easy	Same as PEAP

Table 6.1 The comparison of EAP methods

This thesis implemented MEAP with FreeRADIUS. Here different processes on Linux simulate the client, authenticator and authentication server, which is not practical for the real WLAN environment. An idea-testing environment is to set these three parties (client, authenticator, authentication server) separately. That can make the implementation more reliable.

The management packets, such as de-authentication, disassociation, EAP-Success, EAPOL-Start, or EAPOL-Logoff could be generated to evaluate some types of attacks. Tools for generating arbitrary management packets also could be created by revising or writing the source code of the FreeRADIUS. This thesis gives a simple evaluation about MITM attack, but as of this writing, no publicly available tools implements this MITM attacks on EAP protocols.

References

[1] Jesse Walker, "802.11 Security Series, Part I: The Wired Equivalent Privacy", Intel, pp. 2-3

[2] 802.11i/D10.0, "Draft Amendment to STANDARD for Telecommunications and Information Exchange Between Systems – LAN/MAN Specific Requirements", IEEE Standard, April 2004

[3] 802.1x, "DRAFT Standard for Local and Metropolitan Area Networks—Port-Based Network Access Control (Revision)", IEEE P802.1X-REV/D11, July 22, 2004, pp. 17-29

[4] B. AbobaL, "Extensible Authentication Protocol (EAP)" RFC 3748, June 2004, pp. 22

[5] Ashwin Palekar, "Protected EAP Protocol (PEAP) Version 2, Internet-draft, PPPEXT Working Group, June 2004, pp.11, 68-83

[6] Interlink Networks, Inc. "EAP Methods for Wireless Authentication". April 2, 2003, pp. 7-14

[7] Kwang-Hyun Baek, Sean W.Smith, David Kotz. "A Survey of WPA and 802.11i RSN Authentication Protocols". Technial Report TR2004-524, November 2004, pp. 3-4

[8] Philip Kwan, "802.1x Authentication & Extensible Authentication Protocol (EAP)", White paper, May 2003, pp. 7

[9] Cisco Systems, Inc. "A comprehensive Review of 802.11 Wireless LAN Security and the Cisco Wireless Security Suite", White paper, pp. 31-36

[10] Colonel Donald J.Welch, "A Survey of 802.11a Wireless Security Threats and Security Mechanisms", Technical report, ITOC-TR-2003-101, 2003, pp. 11

[11] Microsoft, "Choosing a Strategy for Wireless LAN Security", white paper, pp. 7-9

[12] C. Rigney, W. Willats, "RADIUS Extensions", RFC2869, June 2000, pp. 11, 21-38

[13] Cisco System, "Authentication with 802.1x and EAP Across Congested WAN Links", Application note, pp. 3

[14] Srivaths Ravi, Securing Wireless Data: System Architecture Challenge

[15] Tomi Hanninen, "Wi-Fi Security", a paper from university of Helsinki, pp. 3

[16] AirDefense, "Wireless LAN Security – What Hackers Know That You Don't", White paper, 2003, pp.2-3

[17] Donald Welch, "Wireless Security Threat Taxonomy", 2003 IEEE Workshop, June 2003, pp. 79-80

[18] T. Dierks, "The TLS Protocol Version 1.0 ", RFC2246, January 1999, pp. 3

[19] B. Aboba, "PPP EAP TLS Authentication Protocol", RFC2716, October 1999, pp. 2

[20] Stanley Wong, "The evolution of wireless security in 802.11 networks: WEP, WPA and 802.11 standards", GSEC Practical v1.4b, May20, 2003, pp. 1-2

[21] Nancy R. Mead, "Wireless Security's Future", IEEE Security & Privacy, 2003, pp. 68-72

[22] Cisco System, "Cisco Response to Dictionary Attacks on Cisco LEAP", Product bulletin, No. 2331, pp. 5

[23] Ken Roser, "HOWTO: EAP/TLS Setup for FreeRADIUS and Windows XP Supplicant", April 2002, pp. 5

[24] Arunesh Mishra, "An Initial Security Analysis of the IEEE 802.1X Standard", CS-TR-4328 UMIACS-TR-2002-10, Feb 2002, pp. 3

[25] Scott Fluhrer, "Weakness in the Key Scheduling Algorithm of RC4", pp. 4

[26] William A. Arbaugh, "Your 802.11 Wireless Network has No Clothes", a paper from University of Maryland, March 2001, pp. 10

[27] Wireless LAN Security FAQ
URL: http://www.drizzle.com/~aboba/IEEE/

[28] Michel Getraide, "Security and authentication for 802.11 wireless networks", a thesis from University of New Orleans, May 2004, pp.27-30

[29] Amleset Kelati, "Application of IEEE 802.1X in HiperLAN type2", a thesis from Chalmers University of Technology, July 2001, pp33-34

[30] Rigney, C. et. al. "Remote Authentication Dial In User Service (RADIUS)". IETF RFC 2865, June, 2000, pp.4

Appendix

1. The install of MIRACL library

1. Unzip the MIRACL.ZIP file using the utility unzip, into an empty directory

```
unzip -j -aa -L miracl.zip
```

The -j ignores the directory structure inside MIRACL.ZIP. The -aa converts all text files to Unix format, and -L ensures that all filenames are lower-case.

2. Perform a tailored build of the MIRACL library by opening an X-Term, and typing

```
bash linux
```

2. The implementation of MEAP method

2.1 eap_speke.h

```
#ifndef _EAP_SPEKE_H
#define _EAP_SPEKE_H

#include "eap.h"
#include "miracl.h"
```

```
#define PW_SPEKE_PRIME                  1
#define PW_SPEKE_INTERMEDIATE           2
#define PW_SPEKE_PROOF                  3
#define PW_SPEKE_RESULT                 4

#define PW_SPEKE_SUCCESS                0
#define PW_SPEKE_FAILURE          1
#define PW_SPEKE_QUERY             2
#define PW_SPEKE_WAITB            3
#define PW_SPEKE_WAITA            4
#define PW_SPEKE_WAITPROOF         5
#define PW_SPEKE_WAITRESULT        6

/*
 ****
 * EAP - SPEKE
 */

/* eap packet structure */
typedef struct speke_handler {
      big prime;
      big random;
      big intermediate;
      big peer;
      big key;

      char proof[20];
```

```
        int state;
  } SPEKE_HANDLER;


/* function declarations here */


SPEKE_HANDLER     *eapspeke_alloc(void);
void              eapspeke_free(SPEKE_HANDLER **packet_ptr);


void speke_init(char* pwd, SPEKE_HANDLER* speke);
int speke_proof(char** id, SPEKE_HANDLER* speke, char* proof, char* test);


int speke_big2tlv(unsigned char type, big num, unsigned char* buf, int size);
//int               eapspeke_compose(EAP_DS *auth, SPEKE_PACKET *packet);
int eapspeke_decode(EAP_DS *auth, SPEKE_HANDLER *packet);
int eapspeke_decodebuf(unsigned char* data, int len, SPEKE_HANDLER *packet);
#endif /*_EAP_SPEKE_H*/
```

2.2 rlm_eap_speke.c

```
/*
 * rlm_eap_speke.c      Handles that are called from eap
 *
 */


#include "autoconf.h"
```

```
#include <stdio.h>
#include <stdlib.h>

#include "eap_speke.h"

#include <rad_assert.h>

miracl *mip = NULL;

static int speke_attach(CONF_SECTION *cs, void **instance)
{
#ifndef MR_NOFULLWIDTH
        mip = mirsys(100,10);
#else
        mip = mirsys(100,10);
#endif

        irand(time(NULL));

        return 0;
}

/*
 *      Initiate the EAP-speke session by sending a challenge to the peer.
 */
static int speke_initiate(void *type_data, EAP_HANDLER *handler)
{
    unsigned char packet[1024];
    unsigned char* ptr = packet;
    char pwd[256];
    int len = 0;

        VALUE_PAIR  *password;
    EAP_DS *eap_ds = handler->eap_ds;

        password = pairfind(handler->request->config_items, PW_PASSWORD);
        if (password == NULL || password->length >= 256) {
                radlog(L_INFO, "rlm_eap_speke: User-Password is required for speke
authentication");
                return 0;
        }

    SPEKE_HANDLER* speke = eapspeke_alloc();
        if (speke == NULL)    {
                radlog(L_ERR, "rlm_eap_speke: out of memory");
                return 0;
        }

    memcpy(pwd, password->strvalue, password->length);
    pwd[password->length] = 0;

    speke_init(pwd, speke);

/*
```

```
     *       Keep track of the challenge.
     */
   handler->opaque = speke;
   handler->free_opaque = free;

   /*
     *       Compose the speke packet out of the data structure,
     *       and free it.
     */
len = speke_big2tlv(PW_SPEKE_PRIME, speke->prime, ptr, 256);
len += speke_big2tlv(PW_SPEKE_INTERMEDIATE, speke->intermediate, ptr + len,
                              256);

eap_ds->request->type.type = PW_EAP_SPEKE;
eap_ds->request->type.data = malloc(len);
if (eap_ds->request->type.data == NULL) {
      radlog(L_ERR, "rlm_eap_speke: out of memory");
      return 0;
}

   ptr = eap_ds->request->type.data;
memcpy(ptr, packet, len);

/* Just the Challenge length */
eap_ds->request->type.length = len;
   eap_ds->request->code = PW_EAP_REQUEST;
```

```
        /*
         *      We don't need to authorize the user at this point.
         *
         *      We also don't need to keep the challenge, as it's
         *      stored in 'handler->eap_ds', which will be given back
         *      to us...
         */
        handler->stage = AUTHENTICATE;
      speke->state = PW_SPEKE_WAITA;

        return 1;
}



/*
 *      Authenticate a previously sent challenge.
 */
static int speke_authenticate(void *arg, EAP_HANDLER *handler)
{
      char test[20];
      char proof[20];
      char* id[] = {"A", "B"};
      unsigned char* ptr;
      SPEKE_HANDLER* speke = (SPEKE_HANDLER*)handler->opaque;
      EAP_DS *eap_ds = handler->eap_ds;

      rad_assert(eap_ds->response->type.type == PW_EAP_SPEKE);
```

```
    rad_assert(handler->request != NULL);
    rad_assert(handler->stage == AUTHENTICATE);

eapspeke_decode(eap_ds, speke);

if(speke->state == PW_SPEKE_WAITA)
{
        speke_proof(id, speke, proof, test);
        if(memcmp(test, speke->proof, 20) != 0)
        {
                radlog(L_ERR, "rlm_eap_speke: wrong proof");
                return 0;
        }

        speke->state = PW_SPEKE_WAITRESULT;

        eap_ds->request->type.type = PW_EAP_SPEKE;
        eap_ds->request->type.data = malloc(22);
        if (eap_ds->request->type.data == NULL) {
                radlog(L_ERR, "rlm_eap_speke: out of memory");
                return 0;
        }
        ptr = eap_ds->request->type.data;

        ptr[0] = PW_SPEKE_PROOF;
        ptr[1] = 22;
        memcpy(ptr + 2, proof, 20);
```

```
            eap_ds->request->type.length = 22;
            eap_ds->request->code = PW_EAP_REQUEST;
        }
        else
        {
                    eap_ds->request->type.length = 0;
            eap_ds->request->code = speke->state;
            radlog(L_ERR, "rlm_eap_speke: state : %d", speke->state);
        }

          /*
           *      Compose the speke packet out of the data structure,
           *      and free it.
           */

          return 1;
}

/*
 *      The module name should be the only globally exported symbol.
 *      That is, everything else should be 'static'.
 */
EAP_TYPE rlm_eap_speke = {
          "eap_speke",
          speke_attach,                           /* attach */
          speke_initiate,                  /* Start the initial request */
        NULL,                       /* authorization */
        speke_authenticate,         /* authentication */
        NULL                        /* detach */
};
```

2.3 eap_speke.c

```
/*
 * eap_speke.c    EAP SPEKE functionality.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "eap.h"

#include "eap_speke.h"

/* large    bit prime p for which (p-1)/2 is also prime */
char *primetext = "1000000000000000000000000000000000000000000000193";

// Hash function
void hash(char *id, char* hash)
{ // hash character string to 160-bit big number
     sha sh;
```

```
        shs_init(&sh);

        while (*id!=0) shs_process(&sh,*id++);

        shs_hash(&sh, hash);

}


int mkstring(char* id, big b1, big b2, char* buf, int len)

{

        int pos = snprintf(buf, len, "%s", id);

        pos += cotstr(b1, buf + pos);

        pos += cotstr(b2, buf + pos);


        return pos;

}


void print_hex(unsigned char* hex, int len)

{

        int i = 0;

        for(i = 0; i < len; i++)

        {

                printf("%02X", hex[i]);

        }

}


/*

 *          Allocate a new SPEKE_HANDLER

 */

SPEKE_HANDLER      *eapspeke_alloc(void)
```

```
{
        SPEKE_HANDLER     *rp;

        if ((rp = malloc(sizeof(SPEKE_HANDLER))) == NULL) {
                radlog(L_ERR, "rlm_eap_speke: out of memory");
                return NULL;
        }

      rp->prime = mirvar(0);
      rp->random = mirvar(0);
      rp->intermediate = mirvar(0);
      rp->peer = mirvar(0);
      rp->key = mirvar(0);
      rp->state = 0;

      cinstr(rp->prime, primetext);

        return rp;
}

/*
 *        Free SPEKE_HANDLER
 */
void            eapspeke_free(SPEKE_HANDLER **packet_ptr)
{
      if(*packet_ptr != NULL)
      {
```

```
                free(*packet_ptr);

                *packet_ptr = NULL;

        }

}


void speke_init(char* pwd, SPEKE_HANDLER* speke)

{

        char hpwd[20];

        big bpwd;

        big num2;

//        big b2rand;


        bpwd = mirvar(0);

        num2 = mirvar(0);

//        b2rand = mirvar(0);


        convert(2, num2);


        hash(pwd, hpwd);

        bytes_to_big(20, hpwd, bpwd);


        bigbits(160, speke->random);

        multiply(speke->random, num2, speke->random);


        powmod(bpwd, speke->random, speke->prime, speke->intermediate);

}
```

```
int speke_proof(char** id, SPEKE_HANDLER* speke, char* proof, char* test)
{
	char buf[4096];

	powmod(speke->peer, speke->random, speke->prime, speke->key);

	if(mkstring(id[0], speke->intermediate, speke->key, buf, 4096) >= 4096)
	{
		radlog(L_ERR, "buf overflow\n");
		return 1;
	}
	hash(buf, proof);

	if(mkstring(id[1], speke->peer, speke->key, buf, 4096) >= 4096)
	{
		radlog(L_ERR, "buf overflow\n");
		return 1;
	}
	hash(buf, test);

	return 0;
}

int speke_big2tlv(unsigned char type, big num, unsigned char* buf, int size)
{
	int len;
	rad_assert(size >= 253);
```

```
    *buf = type;
    len = cotstr(num, (char*)(buf + 2));
//    len = big_to_bytes(253, num, (char*)(buf + 2), FALSE);
    len += 3;
    buf[1] = len;

    return len;
}

/*
int                 eapspeke_compose(EAP_DS *auth, SPEKE_PACKET *packet)
{
}
*/

int eapspeke_decode(EAP_DS *auth, SPEKE_HANDLER *packet)
{
    int len = auth->response->type.length;
    unsigned char* data = auth->response->type.data;

    return eapspeke_decodebuf(data, len, packet);
}

int eapspeke_decodebuf(unsigned char* data, int len, SPEKE_HANDLER* packet)
{
    while(len >= 3)
```

```
	{
		if(*data == PW_SPEKE_PRIME)
		{
			cinstr(packet->prime, (char*)(data + 2));
//			bytes_to_big(data[1] - 2, (char*)(data + 2), packet->prime);
		}
		else if(*data == PW_SPEKE_RESULT)
		{
			packet->state = data[2];
		}
		else if(*data == PW_SPEKE_PROOF)
		{
			memcpy(packet->proof, data + 2, 20);
		}
		else if(*data == PW_SPEKE_INTERMEDIATE)
		{
			cinstr(packet->peer, (char*)(data + 2));
//			bytes_to_big(data[1] - 2, (char*)(data + 2), packet->peer);
		}

		len -= data[1];
		data += data[1];
	}

	return 0;
}
```

2.4 spekeclient.c

```
/*
 * radeapclient.c          EAP specific RADIUS packet debug tool. (Modified based on the
 * program provided by freeRADIUS)
 *
 * Version:     $Id: radeapclient.c,v 1.7.4.5 2006/05/19 14:22:23 nbk Exp $
 *
 *
 *This program is free software; you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation; either version 2 of the License, or
 *    (at your option) any later version.
 *
 *
 * Copyright 2000   The FreeRADIUS server project
 * Copyright 2000   Miquel van Smoorenburg <miquels@cistron.nl>
 * Copyright 2000   Alan DeKok <aland@ox.org>
 */
static const char rcsid[] = "$Id: radeapclient.c,v 1.7.4.5 2006/05/19 14:22:23 nbk Exp $";

#include "autoconf.h"
```

```
#include <stdio.h>
#include <stdlib.h>

#if HAVE_UNISTD_H
#       include <unistd.h>
#endif

#include <string.h>
#include <ctype.h>
#include <netdb.h>
#include <sys/socket.h>

#if HAVE_NETINET_IN_H
#       include <netinet/in.h>
#endif

#if HAVE_SYS_SELECT_H
#       include <sys/select.h>
#endif

#if HAVE_GETOPT_H
#       include <getopt.h>
#endif

#include "conf.h"
#include "radpaths.h"
#include "missing.h"
```

```
#include "../../include/md5.h"


#include "eap_types.h"
#include "eap_sim.h"
#include "eap_speke.h"


miracl *mip;
static SPEKE_HANDLER* speke;


extern int sha1_data_problems;


static int retries = 10;
static float timeout = 3;
static const char *secret = NULL;
static    char *spekepw = NULL;
static int do_output = 1;
static int do_summary = 0;
static int filedone = 0;
static int totalapp = 0;
static int totaldeny = 0;
static char filesecret[256];
const char *RADIUS_dir = RADDBDIR;
const char *progname = "radeapclient";
/* lrad_randctx randctx; */



radlog_dest_t radlog_dest = RADLOG_STDERR;
```

```
const char *radlog_dir = NULL;

int debug_flag = 0;

struct main_config_t mainconfig;

char password[256];


struct eapsim_keys eapsim_mk;


static void NEVER_RETURNS usage(void)

{

        fprintf(stderr, "Usage:  radeapclient  [options]  server[:port]  <command>  <secret>
<password>\n");


        fprintf(stderr, "  <command>      One of auth, acct, status, or disconnect.\n");

        fprintf(stderr, "  -c count       Send each packet 'count' times.\n");

        fprintf(stderr, "  -d raddb       Set dictionary directory.\n");

        fprintf(stderr, "  -f file        Read packets from file, not stdin.\n");

        fprintf(stderr, "  -r retries     If timeout, retry sending the packet 'retries' times.\n");

        fprintf(stderr, "  -t timeout     Wait 'timeout' seconds before retrying (may be a floating
point number).\n");

        fprintf(stderr, "  -i id          Set request id to 'id'.   Values may be 0..255\n");

        fprintf(stderr, "  -S file        read secret from file, not command line.\n");

        fprintf(stderr, "  -q             Do not print anything out.\n");

        fprintf(stderr, "  -s             Print out summary information of auth results.\n");

        fprintf(stderr, "  -v             Show program version information.\n");

        fprintf(stderr, "  -x             Debugging mode.\n");


        exit(1);
```

```
}

int radlog(int lvl, const char *msg, ...)
{
        va_list ap;
        int r;

        r = lvl; /* shut up compiler */

        va_start(ap, msg);
        r = vfprintf(stderr, msg, ap);
        va_end(ap);
        fputc('\n', stderr);

        return r;
}

int log_debug(const char *msg, ...)
{
        va_list ap;
        int r;

        va_start(ap, msg);
        r = vfprintf(stderr, msg, ap);
        va_end(ap);
        fputc('\n', stderr);
```

```
	return r;
}

static int getport(const char *name)
{
	struct	servent		*svp;

	svp = getservbyname (name, "udp");
	if (!svp) {
		return 0;
	}

	return ntohs(svp->s_port);
}

static int send_packet(RADIUS_PACKET *req, RADIUS_PACKET **rep)
{
	int i;
	struct timeval	tv;

	for (i = 0; i < retries; i++) {
		fd_set		rdfdesc;

		rad_send(req, NULL, secret);

		/* And wait for reply, timing out as necessary */
		FD_ZERO(&rdfdesc);
```

```
FD_SET(req->sockfd, &rdfdesc);

tv.tv_sec = (int)timeout;
tv.tv_usec = 1000000 * (timeout - (int) timeout);

/* Something's wrong if we don't get exactly one fd. */
if (select(req->sockfd + 1, &rdfdesc, NULL, NULL, &tv) != 1) {
        continue;
}

*rep = rad_recv(req->sockfd);
if (*rep != NULL) {
        /*
         *      If we get a response from a machine
         *      which we did NOT send a request to,
         *      then complain.
         */
        if (((*rep)->src_ipaddr != req->dst_ipaddr) ||
            ((*rep)->src_port != req->dst_port)) {
                char src[64], dst[64];

                ip_ntoa(src, (*rep)->src_ipaddr);
                ip_ntoa(dst, req->dst_ipaddr);
                fprintf(stderr, "radclient: ERROR: Sent request to host %s port
%d, got response from host %s port %d\n!",
                        dst, req->dst_port,
                        src, (*rep)->src_port);
                        exit(1);
                }
                break;
            } else { /* NULL: couldn't receive the packet */
                librad_perror("radclient:");
                exit(1);
            }
    }

    /* No response or no data read (?) */
    if (i == retries) {
            fprintf(stderr, "radclient: no response from server\n");
            exit(1);
    }

    /*
     *      FIXME: Discard the packet & listen for another.
     *
     *      Hmm... we should really be using eapol_test, which does
     *      a lot more than radeapclient.
     */
    if (rad_verify(*rep, req, secret) != 0) {
            librad_perror("rad_verify");
            exit(1);
    }

    if (rad_decode(*rep, req, secret) != 0) {
```

```
			librad_perror("rad_decode");
			exit(1);
		}

		/* libRADIUS debug already prints out the value pairs for us */
		if (!librad_debug && do_output) {
			printf("Received response ID %d, code %d, length = %d\n",
					(*rep)->id, (*rep)->code, (*rep)->data_len);
			vp_printlist(stdout, (*rep)->vps);
		}
		if((*rep)->code == PW_AUTHENTICATION_ACK) {
			totalapp++;
		} else {
			totaldeny++;
		}

		return 0;
}

static void cleanresp(RADIUS_PACKET *resp)
{
		VALUE_PAIR *vpnext, *vp, **last;


		/*
		 * maybe should just copy things we care about, or keep
		 * a copy of the original input and start from there again?
```

```
         */
        pairdelete(&resp->vps, PW_EAP_MESSAGE);
        pairdelete(&resp->vps, ATTRIBUTE_EAP_BASE+PW_EAP_IDENTITY);

        last = &resp->vps;
        for(vp = *last; vp != NULL; vp = vpnext)
        {
                vpnext = vp->next;

                if((vp->attribute > ATTRIBUTE_EAP_BASE &&
                     vp->attribute <= ATTRIBUTE_EAP_BASE+256) ||
                    (vp->attribute > ATTRIBUTE_EAP_SIM_BASE &&
                     vp->attribute <= ATTRIBUTE_EAP_SIM_BASE+256))
                {
                        *last = vpnext;
                        pairbasicfree(vp);
                } else {
                        last = &vp->next;
                }
        }
}

/*
 * we got an EAP-Request/Sim/Start message in a legal state.
 *
 * pick a supported version, put it into the reply, and insert a nonce.
 */
```

```
}

static int process_eap_start(RADIUS_PACKET *req,
                                    RADIUS_PACKET *rep)
{
        VALUE_PAIR *vp, *newvp;
        VALUE_PAIR *anyidreq_vp, *fullauthidreq_vp, *permanentidreq_vp;
        uint16_t *versions, selectedversion;
        unsigned int i,versioncount;

        /* form new response clear of any EAP stuff */
        cleanresp(rep);

        if((vp                          =                          pairfind(req->vps,
ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_VERSION_LIST)) == NULL) {
                fprintf(stderr, "illegal start message has no VERSION_LIST\n");
                return 0;
        }

        versions = (uint16_t *)vp->strvalue;

        /* verify that the attribute length is big enough for a length field */
        if(vp->length < 4)
        {
                fprintf(stderr, "start message has illegal VERSION_LIST. Too short: %d\n",
vp->length);
                return 0;
        }
        versioncount = ntohs(versions[0])/2;
        /* verify that the attribute length is big enough for the given number
          * of versions present.
          */
        if((unsigned)vp->length <= (versioncount*2 + 2))
        {
                fprintf(stderr, "start message is too short. Claimed %d versions does not fit in %d
bytes\n", versioncount, vp->length);
                return 0;
        }

        /*
          * record the versionlist for the MK calculation.
          */
        eapsim_mk.versionlistlen = versioncount*2;
        memcpy(eapsim_mk.versionlist, (unsigned char *)(versions+1),
                 eapsim_mk.versionlistlen);

        /* walk the version list, and pick the one we support, which
          * at present, is 1, EAP_SIM_VERSION.
          */
        selectedversion=0;
        for(i=0; i < versioncount; i++)
        {
                if(ntohs(versions[i+1]) == EAP_SIM_VERSION)
                {
                        selectedversion=EAP_SIM_VERSION;
                break;
            }
        }
        if(selectedversion == 0)
        {
            fprintf(stderr, "eap-sim start message. No compatible version found. We need
%d\n", EAP_SIM_VERSION);
            for(i=0; i < versioncount; i++)
            {
                fprintf(stderr, "\tfound version %d\n",
                    ntohs(versions[i+1]));
            }
        }

        /*
         * now make sure that we have only FULLAUTH_ID_REQ.
         * I think that it actually might not matter - we can answer in
         * anyway we like, but it is illegal to have more than one
         * present.
         */
        anyidreq_vp                    =                    pairfind(req->vps,
ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_ANY_ID_REQ);
        fullauthidreq_vp               =                    pairfind(req->vps,
ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_FULLAUTH_ID_REQ);
        permanentidreq_vp              =                    pairfind(req->vps,
ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_PERMANENT_ID_REQ);
```

```
        if(fullauthidreq_vp == NULL ||
            anyidreq_vp != NULL ||
            permanentidreq_vp != NULL) {
                fprintf(stderr, "start message has %sanyidreq, %sfullauthid and %spermanentid.
Illegal combination.\n",
                            (anyidreq_vp != NULL ? "a " : "no "),
                            (fullauthidreq_vp != NULL ? "a " : "no "),
                            (permanentidreq_vp != NULL ? "a " : "no "));
                return 0;
        }

        /* okay, we have just any_id_req there, so fill in response */

        /* mark the subtype as being EAP-SIM/Response/Start */
        newvp = paircreate(ATTRIBUTE_EAP_SIM_SUBTYPE, PW_TYPE_INTEGER);
        newvp->lvalue = eapsim_start;
        pairreplace(&(rep->vps), newvp);

        /* insert selected version into response. */
        newvp                                                                              =
paircreate(ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_SELECTED_VERSION,
                                PW_TYPE_OCTETS);
        versions = (uint16_t *)newvp->strvalue;
        versions[0] = htons(selectedversion);
        newvp->length = 2;
        pairreplace(&(rep->vps), newvp);
        /* record the selected version */
        memcpy(eapsim_mk.versionselect, (unsigned char *)versions, 2);


        vp = newvp = NULL;


        {
                uint32_t nonce[4];
                /*
                 * insert a nonce_mt that we make up.
                 */
                newvp                                                          =
    paircreate(ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_NONCE_MT,
                                        PW_TYPE_OCTETS);
                newvp->strvalue[0]=0;
                newvp->strvalue[1]=0;
                newvp->length = 18;    /* 16 bytes of nonce + padding */

                nonce[0]=lrad_rand();
                nonce[1]=lrad_rand();
                nonce[2]=lrad_rand();
                nonce[3]=lrad_rand();
                memcpy(&newvp->strvalue[2], nonce, 16);
                pairreplace(&(rep->vps), newvp);

                /* also keep a copy of the nonce! */
                memcpy(eapsim_mk.nonce_mt, nonce, 16);
        }
        {
                uint16_t *pidlen, idlen;

                /*
                 * insert the identity here.
                 */
                vp = pairfind(rep->vps, PW_USER_NAME);
                if(vp == NULL)
                {
                        fprintf(stderr, "eap-sim: We need to have a User-Name attribute!\n");
                        return 0;
                }
                newvp                                                      =
paircreate(ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_IDENTITY,
                                        PW_TYPE_OCTETS);
                idlen = strlen((char*)vp->strvalue);
                pidlen = (uint16_t *)newvp->strvalue;
                *pidlen = htons(idlen);
                newvp->length = idlen + 2;

                memcpy(&newvp->strvalue[2], vp->strvalue, idlen);
                pairreplace(&(rep->vps), newvp);

                /* record it */
                memcpy(eapsim_mk.identity, vp->strvalue, idlen);
                eapsim_mk.identitylen = idlen;
        }

        return 1;
}

/*
 * we got an EAP-Request/Sim/Challenge message in a legal state.
 *
 * use the RAND challenge to produce the SRES result, and then
 * use that to generate a new MAC.
 *
 * for the moment, we ignore the RANDs, then just plug in the SRES
 * values.
 *
 */
static int process_eap_challenge(RADIUS_PACKET *req,
                                 RADIUS_PACKET *rep)
{
        VALUE_PAIR *newvp;
        VALUE_PAIR *mac, *randvp;
        VALUE_PAIR *sres1,*sres2,*sres3;
        VALUE_PAIR *Kc1, *Kc2, *Kc3;
        uint8_t calcmac[20];

        /* look for the AT_MAC and the challenge data */
        mac     = pairfind(req->vps, ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_MAC);
        randvp= pairfind(req->vps, ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_RAND);
	if(mac == NULL || rand == NULL) {
		fprintf(stderr, "radeapclient: challenge message needs to contain RAND and
MAC\n");
		return 0;
	}

	/*
	 * compare RAND with randX, to verify this is the right response
	 * to this challenge.
	 */
	{
	  VALUE_PAIR *randcfgvp[3];
	  unsigned char *randcfg[3];

	  randcfg[0] = &randvp->strvalue[2];
	  randcfg[1] = &randvp->strvalue[2+EAPSIM_RAND_SIZE];
	  randcfg[2] = &randvp->strvalue[2+EAPSIM_RAND_SIZE*2];

	  randcfgvp[0] = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_RAND1);
	  randcfgvp[1] = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_RAND2);
	  randcfgvp[2] = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_RAND3);

	  if(randcfgvp[0] == NULL ||
	     randcfgvp[1] == NULL ||
	     randcfgvp[2] == NULL) {
	    fprintf(stderr, "radeapclient: needs to have rand1, 2 and 3 set.\n");
	    return 0;
```

```
}

if(memcmp(randcfg[0], randcfgvp[0]->strvalue, EAPSIM_RAND_SIZE)!=0 ||
   memcmp(randcfg[1], randcfgvp[1]->strvalue, EAPSIM_RAND_SIZE)!=0 ||
   memcmp(randcfg[2], randcfgvp[2]->strvalue, EAPSIM_RAND_SIZE)!=0) {
  int rnum,i,j;

  fprintf(stderr, "radeapclient: one of rand 1,2,3 didn't match\n");
  for(rnum = 0; rnum < 3; rnum++) {
    fprintf(stderr, "received     rand %d: ", rnum);
    j=0;
    for (i = 0; i < EAPSIM_RAND_SIZE; i++) {
     if(j==4) {
       printf("_");
       j=0;
     }
     j++;

     fprintf(stderr, "%02x", randcfg[rnum][i]);
    }
    fprintf(stderr, "\nconfigured rand %d: ", rnum);
    j=0;
    for (i = 0; i < EAPSIM_RAND_SIZE; i++) {
     if(j==4) {
       printf("_");
       j=0;
     }
```

```
            j++;


            fprintf(stderr, "%02x", randcfgvp[rnum]->strvalue[i]);
          }
          fprintf(stderr, "\n");
        }
        return 0;
      }
    }


    /*
     * now dig up the sres values from the response packet,
     * which were put there when we read things in.
     *
     * Really, they should be calculated from the RAND!
     *
     */
    sres1 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_SRES1);
    sres2 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_SRES2);
    sres3 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_SRES3);


    if(sres1 == NULL ||
       sres2 == NULL ||
       sres3 == NULL) {
            fprintf(stderr, "radeapclient: needs to have sres1, 2 and 3 set.\n");
            return 0;
    }
```

```
memcpy(eapsim_mk.sres[0], sres1->strvalue, sizeof(eapsim_mk.sres[0]));
memcpy(eapsim_mk.sres[1], sres2->strvalue, sizeof(eapsim_mk.sres[1]));
memcpy(eapsim_mk.sres[2], sres3->strvalue, sizeof(eapsim_mk.sres[2]));

Kc1 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_KC1);
Kc2 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_KC2);
Kc3 = pairfind(rep->vps, ATTRIBUTE_EAP_SIM_KC3);

if(Kc1 == NULL ||
    Kc2 == NULL ||
    Kc3 == NULL) {
        fprintf(stderr, "radeapclient: needs to have Kc1, 2 and 3 set.\n");
        return 0;
}
memcpy(eapsim_mk.Kc[0], Kc1->strvalue, sizeof(eapsim_mk.Kc[0]));
memcpy(eapsim_mk.Kc[1], Kc2->strvalue, sizeof(eapsim_mk.Kc[1]));
memcpy(eapsim_mk.Kc[2], Kc3->strvalue, sizeof(eapsim_mk.Kc[2]));

/* all set, calculate keys */
eapsim_calculate_keys(&eapsim_mk);

if(debug_flag) {
   eapsim_dump_mk(&eapsim_mk);
}

/* verify the MAC, now that we have all the keys. */
if(eapsim_checkmac(req->vps, eapsim_mk.K_aut,
                        eapsim_mk.nonce_mt, sizeof(eapsim_mk.nonce_mt),
                        calcmac)) {
            printf("MAC check succeed\n");
    } else {
            int i, j;
            j=0;
            printf("calculated MAC (");
            for (i = 0; i < 20; i++) {
                    if(j==4) {
                            printf("_");
                            j=0;
                    }
                    j++;

                    printf("%02x", calcmac[i]);
            }
            printf(" did not match\n");
            return 0;
    }

/* form new response clear of any EAP stuff */
cleanresp(rep);

/* mark the subtype as being EAP-SIM/Response/Start */
newvp = paircreate(ATTRIBUTE_EAP_SIM_SUBTYPE, PW_TYPE_INTEGER);
newvp->lvalue = eapsim_challenge;
pairreplace(&(rep->vps), newvp);
```

```
        /*
         * fill the SIM_MAC with a field that will in fact get appended
         * to the packet before the MAC is calculated
         */
        newvp = paircreate(ATTRIBUTE_EAP_SIM_BASE+PW_EAP_SIM_MAC,
                           PW_TYPE_OCTETS);
        memcpy(newvp->strvalue+EAPSIM_SRES_SIZE*0,                    sres1->strvalue,
EAPSIM_SRES_SIZE);
        memcpy(newvp->strvalue+EAPSIM_SRES_SIZE*1,                    sres2->strvalue,
EAPSIM_SRES_SIZE);
        memcpy(newvp->strvalue+EAPSIM_SRES_SIZE*2,                    sres3->strvalue,
EAPSIM_SRES_SIZE);
        newvp->length = EAPSIM_SRES_SIZE*3;
        pairreplace(&(rep->vps), newvp);

        newvp = paircreate(ATTRIBUTE_EAP_SIM_KEY, PW_TYPE_OCTETS);
        memcpy(newvp->strvalue,     eapsim_mk.K_aut, EAPSIM_AUTH_SIZE);
        newvp->length = EAPSIM_AUTH_SIZE;
        pairreplace(&(rep->vps), newvp);

        return 1;
}

/*
 * this code runs the EAP-SIM client state machine.
 * the *request* is from the server.
 * the *reponse* is to the server.
 *
 */
static int respond_eap_sim(RADIUS_PACKET *req,
                                    RADIUS_PACKET *resp)
{
          enum eapsim_clientstates state, newstate;
          enum eapsim_subtype subtype;
          VALUE_PAIR *vp, *statevp, *radstate, *eapid;
          char statenamebuf[32], subtypenamebuf[32];

          if ((radstate = paircopy2(req->vps, PW_STATE)) == NULL)
          {
                    return 0;
          }

          if ((eapid = paircopy2(req->vps, ATTRIBUTE_EAP_ID)) == NULL)
          {
                    return 0;
          }

          /* first, dig up the state from the request packet, setting
            * outselves to be in EAP-SIM-Start state if there is none.
            */

          if((statevp = pairfind(resp->vps, ATTRIBUTE_EAP_SIM_STATE)) == NULL)
          {
```

```
		/* must be initial request */
		statevp = paircreate(ATTRIBUTE_EAP_SIM_STATE, PW_TYPE_INTEGER);
		statevp->lvalue = eapsim_client_init;
		pairreplace(&(resp->vps), statevp);
	}
	state = statevp->lvalue;

	/*
	 * map the attributes, and authenticate them.
	 */
	unmap_eapsim_types(req);

	printf("<+++ EAP-sim decoded packet:\n");
	vp_printlist(stdout, req->vps);

	if((vp = pairfind(req->vps, ATTRIBUTE_EAP_SIM_SUBTYPE)) == NULL)
	{
		return 0;
	}
	subtype = vp->lvalue;

	/*
	 * look for the appropriate state, and process incoming message
	 */
	switch(state) {
	case eapsim_client_init:
		switch(subtype) {
```

```
                case eapsim_start:
                        newstate = process_eap_start(req, resp);
                        break;

                case eapsim_challenge:
                case eapsim_notification:
                case eapsim_reauth:
                default:
                        fprintf(stderr, "radeapclient: sim in state %s message %s is illegal. Reply
dropped.\n",
                                sim_state2name(state, statenamebuf, sizeof(statenamebuf)),
                                sim_subtype2name(subtype,                    subtypenamebuf,
sizeof(subtypenamebuf)));
                        /* invalid state, drop message */
                        return 0;
                }
                break;

        case eapsim_client_start:
                switch(subtype) {
                case eapsim_start:
                        /* NOT SURE ABOUT THIS ONE, retransmit, I guess */
                        newstate = process_eap_start(req, resp);
                        break;

                case eapsim_challenge:
                        newstate = process_eap_challenge(req, resp);
```

```
				break;


			default:
				fprintf(stderr, "radeapclient: sim in state %s message %s is illegal. Reply
dropped.\n",
					sim_state2name(state, statenamebuf, sizeof(statenamebuf)),
					sim_subtype2name(subtype,                    subtypenamebuf,
sizeof(subtypenamebuf)));
				/* invalid state, drop message */
				return 0;
			}
			break;



		default:
			fprintf(stderr, "radeapclient: sim in illegal state %s\n",
				sim_state2name(state, statenamebuf, sizeof(statenamebuf)));
			return 0;
		}


		/* copy the eap state object in */
		pairreplace(&(resp->vps), eapid);


		/* update stete info, and send new packet */
		map_eapsim_types(resp);


		/* copy the RADIUS state object in */
```

```
        pairreplace(&(resp->vps), radstate);


        statevp->lvalue = newstate;
        return 1;
}


static int respond_eap_speke(RADIUS_PACKET *req,
                             RADIUS_PACKET *rep)
{
        VALUE_PAIR *vp, *id, *state;
        unsigned char identifier;
//      char    response[16];


      char proof[20];
      char test[20];
      char* proofid[] = {"B", "A"};


        cleanresp(rep);


        if ((state = paircopy2(req->vps, PW_STATE)) == NULL)
        {
                fprintf(stderr, "radeapclient: no state attribute found\n");
                return 0;
        }


        if ((id = paircopy2(req->vps, ATTRIBUTE_EAP_ID)) == NULL)
        {
                fprintf(stderr, "radeapclient: no EAP-ID attribute found\n");
                return 0;
        }
        identifier = id->lvalue;

        if ((vp = pairfind(req->vps, ATTRIBUTE_EAP_BASE+PW_EAP_SPEKE)) == NULL)
        {
                fprintf(stderr, "radeapclient: no EAP-MD5 attribute found\n");
                return 0;
        }

    eapspeke_decodebuf(vp->strvalue, vp->length, speke);

        vp = paircreate(ATTRIBUTE_EAP_BASE+PW_EAP_SPEKE, PW_TYPE_OCTETS);

    if(speke->state == PW_SPEKE_WAITB)
    {
          speke_init(spekepw, speke);

          speke_proof(proofid, speke, proof, test);

          vp->length = speke_big2tlv(PW_SPEKE_INTERMEDIATE, speke->intermediate,
                                              vp->strvalue, 253);
          vp->strvalue[vp->length] = PW_SPEKE_PROOF;
          vp->strvalue[vp->length + 1] = 22;
          memcpy(vp->strvalue + vp->length + 2, proof, 20);
          vp->length += 22;
        speke->state = PW_SPEKE_WAITPROOF;
    }
    else if(speke->state == PW_SPEKE_WAITPROOF)
    {
        vp->length = 3;
        vp->strvalue[0] = PW_SPEKE_RESULT;
        vp->strvalue[1] = 3;

        speke_proof(proofid, speke, proof, test);
        if(memcmp(speke->proof, test, 20) == 0)
        {
            vp->strvalue[2] = PW_EAP_SUCCESS;
            printf("speke success\n");
        }
        else
        {
            vp->strvalue[2] = PW_EAP_FAILURE;
            printf("speke failed\n");
        }
    }
    else
    {
        return 0;
    }

      pairreplace(&(rep->vps), vp);
```

```
        pairreplace(&(rep->vps), id);


        /* copy the state object in */
        pairreplace(&(rep->vps), state);


        return 1;
}

static int respond_eap_md5(RADIUS_PACKET *req,
                           RADIUS_PACKET *rep)
{
        VALUE_PAIR *vp, *id, *state;
        int valuesize, namesize;
        unsigned char identifier;
        unsigned char *value;
        unsigned char *name;
        MD5_CTX     context;
        char      response[16];

        cleanresp(rep);

        if ((state = paircopy2(req->vps, PW_STATE)) == NULL)
        {
                fprintf(stderr, "radeapclient: no state attribute found\n");
                return 0;
        }
```

```
if ((id = paircopy2(req->vps, ATTRIBUTE_EAP_ID)) == NULL)
{
        fprintf(stderr, "radeapclient: no EAP-ID attribute found\n");
        return 0;
}
identifier = id->lvalue;

if ((vp = pairfind(req->vps, ATTRIBUTE_EAP_BASE+PW_EAP_MD5)) == NULL)
{
        fprintf(stderr, "radeapclient: no EAP-MD5 attribute found\n");
        return 0;
}

/* got the details of the MD5 challenge */
valuesize = vp->strvalue[0];
value = &vp->strvalue[1];
name    = &vp->strvalue[valuesize+1];
namesize = vp->length - (valuesize + 1);

/* sanitize items */
if(valuesize > vp->length)
{
        fprintf(stderr, "radeapclient: md5 valuesize if too big (%d > %d)\n",
                valuesize, vp->length);
        return 0;
}
```

```
        /* now do the CHAP operation ourself, rather than build the
         * buffer. We could also call rad_chap_encode, but it wants
         * a CHAP-Challenge, which we don't want to bother with.
         */
        librad_MD5Init(&context);
        librad_MD5Update(&context, &identifier, 1);
//      librad_MD5Update(&context, password, strlen(password));
//      librad_MD5Update(&context, value, valuesize);
//      librad_MD5Final(response, &context);

        vp = paircreate(ATTRIBUTE_EAP_BASE+PW_EAP_MD5, PW_TYPE_OCTETS);
        vp->strvalue[0]=16;
        memcpy(&vp->strvalue[1], response, 16);
        vp->length = 17;

        pairreplace(&(rep->vps), vp);

        pairreplace(&(rep->vps), id);

        /* copy the state object in */
        pairreplace(&(rep->vps), state);

        return 1;
}
```

```
static int sendrecv_eap(RADIUS_PACKET *rep)
{
        RADIUS_PACKET *req = NULL;
        VALUE_PAIR *vp, *vpnext;
        int tried_eap_md5 = 0;

        /*
         *      Keep a copy of the the User-Password attribute.
         */
        if ((vp = pairfind(rep->vps, ATTRIBUTE_EAP_MD5_PASSWORD)) != NULL) {
                strNcpy(password, (char *)vp->strvalue, sizeof(vp->strvalue));

        } else  if ((vp = pairfind(rep->vps, PW_PASSWORD)) != NULL) {
                strNcpy(password, (char *)vp->strvalue, sizeof(vp->strvalue));
                /*
                 *      Otherwise keep a copy of the CHAP-Password attribute.
                 */
        } else if ((vp = pairfind(rep->vps, PW_CHAP_PASSWORD)) != NULL) {
                strNcpy(password, (char *)vp->strvalue, sizeof(vp->strvalue));
        } else {
                *password = '\0';
        }

 again:
        rep->id++;

        printf("\n+++> About to send encoded packet:\n");
```

```
vp_printlist(stdout, rep->vps);


/*
 * if there are EAP types, encode them into an EAP-Message
 *
 */
map_eap_types(rep);


/*
 *  Fix up Digest-Attributes issues
 */
for (vp = rep->vps; vp != NULL; vp = vp->next) {
        switch (vp->attribute) {
        default:
                break;


        case PW_DIGEST_REALM:
        case PW_DIGEST_NONCE:
        case PW_DIGEST_METHOD:
        case PW_DIGEST_URI:
        case PW_DIGEST_QOP:
        case PW_DIGEST_ALGORITHM:
        case PW_DIGEST_BODY_DIGEST:
        case PW_DIGEST_CNONCE:
        case PW_DIGEST_NONCE_COUNT:
        case PW_DIGEST_USER_NAME:
                /* overlapping! */
```

```
				memmove(&vp->strvalue[2], &vp->strvalue[0], vp->length);
				vp->strvalue[0] = vp->attribute - PW_DIGEST_REALM + 1;
				vp->length += 2;
				vp->strvalue[1] = vp->length;
				vp->attribute = PW_DIGEST_ATTRIBUTES;
				break;
		}
	}

	/*
	 *	If we've already sent a packet, free up the old
	 *	one, and ensure that the next packet has a unique
	 *	ID and authentication vector.
	 */
	if (rep->data) {
		free(rep->data);
		rep->data = NULL;
	}

	librad_md5_calc(rep->vector, rep->vector,
				sizeof(rep->vector));

	if (*password != '\0') {
		if ((vp = pairfind(rep->vps, PW_PASSWORD)) != NULL) {
			strNcpy((char *)vp->strvalue, password, strlen(password) + 1);
			vp->length = strlen(password);
```

```
        } else if ((vp = pairfind(rep->vps, PW_CHAP_PASSWORD)) != NULL) {
                strNcpy((char *)vp->strvalue, password, strlen(password) + 1);
                vp->length = strlen(password);

                rad_chap_encode(rep, (char *) vp->strvalue, rep->id, vp);
                vp->length = 17;
        }
} /* there WAS a password */

/* send the response, wait for the next request */
send_packet(rep, &req);

/* okay got back the packet, go and decode the EAP-Message. */
unmap_eap_types(req);

printf("<+++ EAP decoded packet:\n");
vp_printlist(stdout, req->vps);

/* now look for the code type. */
for (vp = req->vps; vp != NULL; vp = vpnext) {
        vpnext = vp->next;

        switch (vp->attribute) {
        default:
                break;

        case ATTRIBUTE_EAP_BASE+PW_EAP_MD5:
```

```
                    if(respond_eap_md5(req, rep) && tried_eap_md5 < 3)
                    {
                            tried_eap_md5++;
                            goto again;
                    }
                    break;

                case ATTRIBUTE_EAP_BASE+PW_EAP_SIM:
                    if(respond_eap_sim(req, rep))
                    {
                            goto again;
                    }
                    break;

            case ATTRIBUTE_EAP_BASE+PW_EAP_SPEKE:
                if(respond_eap_speke(req, rep))
                {
                    goto again;
                }
                break;

                }
        }

        return 1;
}
```

```
int main(int argc, char **argv)

{

        RADIUS_PACKET *req;

        char *p;

        int c;

        int port = 0;

        char *filename = NULL;

        FILE *fp;

        int count = 1;

        int id;


        id = ((int)getpid() & 0xff);

        librad_debug = 0;


        radlog_dest = RADLOG_STDERR;


        while ((c = getopt(argc, argv, "c:d:f:hi:qst:r:S:xXv")) != EOF)

        {

                switch(c) {

                case 'c':

                        if (!isdigit((int) *optarg))

                                usage();

                        count = atoi(optarg);

                        break;

                case 'd':

                        RADIUS_dir = optarg;
                break;
        case 'f':
                filename = optarg;
                break;
        case 'q':
                do_output = 0;
                break;
        case 'x':
        debug_flag++;
                librad_debug++;
                break;

        case 'X':
#if 0
        sha1_data_problems = 1; /* for debugging only */
#endif
        break;



        case 'r':
                if (!isdigit((int) *optarg))
                        usage();
                retries = atoi(optarg);
                break;
        case 'i':
                if (!isdigit((int) *optarg))
```

```
                        usage();
                id = atoi(optarg);
                if ((id < 0) || (id > 255)) {
                        usage();
                }
                break;
        case 's':
                do_summary = 1;
                break;
        case 't':
                if (!isdigit((int) *optarg))
                        usage();
                timeout = atof(optarg);
                break;
        case 'v':
                printf("radclient: $Id: radeapclient.c,v 1.7.4.5 2006/05/19 14:22:23 nbk
Exp $ built on " __DATE__ " at " __TIME__ "\n");
                exit(0);
                break;
    case 'S':
        fp = fopen(optarg, "r");
        if (!fp) {
            fprintf(stderr, "radclient: Error opening %s: %s\n",
                    optarg, strerror(errno));
            exit(1);
        }
        if (fgets(filesecret, sizeof(filesecret), fp) == NULL) {
            fprintf(stderr, "radclient: Error reading %s: %s\n",
                        optarg, strerror(errno));
            exit(1);
        }
        fclose(fp);

        /* truncate newline */
        p = filesecret + strlen(filesecret) - 1;
        while ((p >= filesecret) &&
                  (*p < ' ')) {
            *p = '\0';
            --p;
        }

        if (strlen(filesecret) < 2) {
            fprintf(stderr, "radclient: Secret in %s is too short\n", optarg);
            exit(1);
        }
        secret = filesecret;
        break;
         case 'h':
         default:
                  usage();
                  break;
         }
}
argc -= (optind - 1);
```

```
	argv += (optind - 1);


	if ((argc < 4)    ||
	      ((secret == NULL) && (argc < 5))) {
		usage();
	}


	if (dict_init(RADIUS_dir, RADIUS_DICTIONARY) < 0) {
		librad_perror("radclient");
		return 1;
	}


	if ((req = rad_alloc(1)) == NULL) {
		librad_perror("radclient");
		exit(1);
	}


#if 0
	{
		FILE *randinit;


		if((randinit = fopen("/dev/urandom", "r")) == NULL)
		{
			perror("/dev/urandom");
		} else {
			fread(randctx.randrsl, 256, 1, randinit);
			fclose(randinit);
		}
	}
	lrad_randinit(&randctx, 1);
#endif


	req->id = id;


	/*
	 *	Strip port from hostname if needed.
	 */
	if ((p = strchr(argv[1], ':')) != NULL) {
		*p++ = 0;
		port = atoi(p);
	}


	/*
	 *	See what kind of request we want to send.
	 */
	if (strcmp(argv[2], "auth") == 0) {
		if (port == 0) port = getport("RADIUS");
		if (port == 0) port = PW_AUTH_UDP_PORT;
		req->code = PW_AUTHENTICATION_REQUEST;

	} else if (strcmp(argv[2], "acct") == 0) {
		if (port == 0) port = getport("radacct");
		if (port == 0) port = PW_ACCT_UDP_PORT;
		req->code = PW_ACCOUNTING_REQUEST;
		do_summary = 0;


	} else if (strcmp(argv[2], "status") == 0) {
		if (port == 0) port = getport("RADIUS");
		if (port == 0) port = PW_AUTH_UDP_PORT;
		req->code = PW_STATUS_SERVER;


	} else if (strcmp(argv[2], "disconnect") == 0) {
		if (port == 0) port = PW_POD_UDP_PORT;
		req->code = PW_DISCONNECT_REQUEST;


	} else if (isdigit((int) argv[2][0])) {
		if (port == 0) port = getport("RADIUS");
		if (port == 0) port = PW_AUTH_UDP_PORT;
		req->code = atoi(argv[2]);
	} else {
		usage();
	}


	/*
	 *	Ensure that the configuration is initialized.
	 */
	memset(&mainconfig, 0, sizeof(mainconfig));


	/*
	 *	Resolve hostname.
	 */
```

```
    req->dst_port = port;
    req->dst_ipaddr = ip_getaddr(argv[1]);
    if (req->dst_ipaddr == INADDR_NONE) {
            fprintf(stderr, "radclient: Failed to find IP address for host %s\n", argv[1]);
            exit(1);
    }


    /*
     *      Add the secret.
     */
    if (argv[3]) secret = argv[3];
if(argv[4]) spekepw = argv[4];



    /*
     *      Read valuepairs.
     *      Maybe read them, from stdin, if there's no
     *      filename, or if the filename is '-'.
     */
    if (filename && (strcmp(filename, "-") != 0)) {
            fp = fopen(filename, "r");
            if (!fp) {
                    fprintf(stderr, "radclient: Error opening %s: %s\n",
                     filename, strerror(errno));
                    exit(1);
            }
    } else {
```

```
                fp = stdin;
        }

#ifndef MR_NOFULLWIDTH
    mip = mirsys(100,0);
#else
    mip = mirsys(100,MAXBASE);
#endif

/*
    big bpwd;
    big num2;
    bpwd = mirvar(0);
    num2 = mirvar(0);

    cinstr(num2,
            "99999999999999999999999999999999999999999999999");
    nxprime(num2, bpwd);

    cotnum(bpwd, stdout);
    cotnum(num2, stdout);
*/

    irand(time(NULL));
    speke = eapspeke_alloc();
    speke->state = PW_SPEKE_WAITB;
```

```
/*      SPEKE_HANDLER* speke2 = eapspeke_alloc();


     speke_init("testing", speke);

     speke_init("testing", speke2);


     powmod(speke->intermediate, speke2->random, speke2->prime, speke2->key);


     printf("Ka:");

     cotnum(speke->key, stdout);

     printf("Kb:");

     cotnum(speke2->key, stdout);

*/

//      return 0;


        /*

         *      Send request.

         */

        if ((req->sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {

                perror("radclient: socket: ");

                exit(1);

        }


        while(!filedone) {

                if(req->vps) pairfree(&req->vps);


                if ((req->vps = readvp2(fp, &filedone, "radeapclient:"))

                      == NULL) {
```

```
                break;
            }

            sendrecv_eap(req);
        }

        if(do_summary) {
            printf("\n\t    Total approved auths:   %d\n", totalapp);
            printf("\t      Total denied auths:   %d\n", totaldeny);
        }
        return 0;
}
```

3. The output from RADIUS server

3.1. The successful authentication:

```
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=217, length=75
        User-Name = "testuser"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 0
        EAP-Message = 0x02d2000d0174657374757365572
        Message-Authenticator = 0xd35eed63602313f202a9ec1c72ddb275
  Processing the authorize section of RADIUSd.conf
modcall: entering group authorize for request 14
  modcall[authorize]: module "preprocess" returns ok for request 14
  modcall[authorize]: module "chap" returns noop for request 14
  modcall[authorize]: module "mschap" returns noop for request 14
    rlm_realm: No '@' in User-Name = "testuser", looking up realm NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 14
  rlm_eap: EAP packet type response id 210 length 13
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 14
    users: Matched entry testuser at line 80
  modcall[authorize]: module "files" returns ok for request 14
modcall: leaving group authorize (returns updated) for request 14
  rad_check_password:   Found Auth-Type EAP
auth: type "EAP"
  Processing the authenticate section of RADIUSd.conf
modcall: entering group authenticate for request 14
  rlm_eap: EAP Identity
  rlm_eap: processing type speke
  modcall[authenticate]: module "eap" returns handled for request 14
modcall: leaving group authenticate (returns handled) for request 14
Sending Access-Challenge of id 217 to 127.0.0.1 port 32770
            EAP-Message                                                    =
0x01d3006c18013431303030303030303030303030303030303030303030303030303030303030
30303030303030303030303030303030303030303030313933300023333336373936373037303131
36373231323133303335313333393835343935333339393138373239333330363833333730
00          Message-Authenticator = 0x00000000000000000000000000000000
            State = 0x5045db9b095ddc9fbda179f7625d5e73
Finished request 14
```

```
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=218, length=158
            User-Name = "testuser"
            NAS-IP-Address = 127.0.0.1
            NAS-Port = 0
            Message-Authenticator = 0xb9b82e11dec137df5c46ddf4faf96ea4
            State = 0x5045db9b095ddc9fbda179f7625d5e73
            EAP-Message                                                            =
0x02d3004e18023338303635383536353131333034383437333539333238383731363133
3532333235303539323737323036373930343735000316a0de1f4e8d7e48f219cbdd04d83
090f6342da572
  Processing the authorize section of RADIUSd.conf
modcall: entering group authorize for request 15
  modcall[authorize]: module "preprocess" returns ok for request 15
  modcall[authorize]: module "chap" returns noop for request 15
  modcall[authorize]: module "mschap" returns noop for request 15
    rlm_realm: No '@' in User-Name = "testuser", looking up realm NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 15
  rlm_eap: EAP packet type response id 211 length 78
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 15
    users: Matched entry testuser at line 80
  modcall[authorize]: module "files" returns ok for request 15
modcall: leaving group authorize (returns updated) for request 15
```

```
  rad_check_password:   Found Auth-Type EAP
auth: type "EAP"
  Processing the authenticate section of RADIUSd.conf
modcall: entering group authenticate for request 15
  rlm_eap: Request found, released from the list
  rlm_eap: EAP/speke
  rlm_eap: processing type speke
  modcall[authenticate]: module "eap" returns handled for request 15
modcall: leaving group authenticate (returns handled) for request 15
Sending Access-Challenge of id 218 to 127.0.0.1 port 32770
        EAP-Message                                                      =
0x01d4001b18031665f401d17ed89e4640068fe9d8e9edac99359cac
        Message-Authenticator = 0x00000000000000000000000000000000
        State = 0xc81a43ac5afe1a4767d97d8a118c7507
Finished request 15
Going to the next request
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=219, length=88
        User-Name = "testuser"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 0
        Message-Authenticator = 0x132b75ba4114af8b228ef7cb9614af63
        State = 0xc81a43ac5afe1a4767d97d8a118c7507
        EAP-Message = 0x02d4000818040303
  Processing the authorize section of RADIUSd.conf
modcall: entering group authorize for request 16
  modcall[authorize]: module "preprocess" returns ok for request 16
  modcall[authorize]: module "chap" returns noop for request 16
  modcall[authorize]: module "mschap" returns noop for request 16
    rlm_realm: No '@' in User-Name = "testuser", looking up realm NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 16
  rlm_eap: EAP packet type response id 212 length 8
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 16
    users: Matched entry testuser at line 80
  modcall[authorize]: module "files" returns ok for request 16
modcall: leaving group authorize (returns updated) for request 16
  rad_check_password:   Found Auth-Type EAP
auth: type "EAP"
  Processing the authenticate section of RADIUSd.conf
modcall: entering group authenticate for request 16
  rlm_eap: Request found, released from the list
  rlm_eap: EAP/speke
  rlm_eap: processing type speke
rlm_eap_speke: state : 3
  rlm_eap: Freeing handler
  modcall[authenticate]: module "eap" returns ok for request 16
modcall: leaving group authenticate (returns ok) for request 16
Sending Access-Accept of id 219 to 127.0.0.1 port 32770
        EAP-Message = 0x03d40004
        Message-Authenticator = 0x00000000000000000000000000000000
        User-Name = "testuser"
Finished request 16
```

```
Going to the next request
Waking up in 6 seconds...
--- Walking the entire request list ---
Cleaning up request 14 ID 217 with timestamp 44f51806
Cleaning up request 15 ID 218 with timestamp 44f51806
Cleaning up request 16 ID 219 with timestamp 44f51806
Nothing to do.  Sleeping until we see a request.
```

3.2. The failed wrong shared secret authentication

```
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Received packet from 127.0.0.1 with invalid Message-Authenticator!  (Shared secret is
incorrect.) Dropping packet without response.
Finished request 17
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
Waking up in 3 seconds...
--- Walking the entire request list ---
Cleaning up request 17 ID 37 with timestamp 44f51a3e
Nothing to do.  Sleeping until we see a request.
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Received packet from 127.0.0.1 with invalid Message-Authenticator!  (Shared secret is
incorrect.) Dropping packet without response.
```

```
Finished request 18
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
Waking up in 3 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
Cleaning up request 18 ID 37 with timestamp 44f51a44
Nothing to do.  Sleeping until we see a request.
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Received packet from 127.0.0.1 with invalid Message-Authenticator!  (Shared secret is
incorrect.) Dropping packet without response.
Finished request 19
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
Waking up in 3 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
```

```
Cleaning up request 19 ID 37 with timestamp 44f51a4d
Nothing to do.  Sleeping until we see a request.
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Received packet from 127.0.0.1 with invalid Message-Authenticator!  (Shared secret is
incorrect.) Dropping packet without response.
Finished request 20
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=37, length=75
Discarding duplicate request from client localhost:32770 - ID: 37
--- Walking the entire request list ---
Waking up in 3 seconds...
--- Walking the entire request list ---
Cleaning up request 20 ID 37 with timestamp 44f51a56
Nothing to do.  Sleeping until we see a request.
```

3.3. The failed wrong password authentication

```
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=122, length=75
        User-Name = "testuser"
        NAS-IP-Address = 127.0.0.1
        NAS-Port = 0
        EAP-Message = 0x02d2000d0174657374757365572
        Message-Authenticator = 0xf099f1429405400efe2e2fd0a3d2f919
  Processing the authorize section of RADIUSd.conf
```

```
modcall: entering group authorize for request 21
  modcall[authorize]: module "preprocess" returns ok for request 21
  modcall[authorize]: module "chap" returns noop for request 21
  modcall[authorize]: module "mschap" returns noop for request 21
    rlm_realm: No '@' in User-Name = "testuser", looking up realm NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 21
  rlm_eap: EAP packet type response id 210 length 13
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 21
    users: Matched entry testuser at line 80
  modcall[authorize]: module "files" returns ok for request 21
modcall: leaving group authorize (returns updated) for request 21
  rad_check_password:   Found Auth-Type EAP
auth: type "EAP"
  Processing the authenticate section of RADIUSd.conf
modcall: entering group authenticate for request 21
  rlm_eap: EAP Identity
  rlm_eap: processing type speke
  modcall[authenticate]: module "eap" returns handled for request 21
modcall: leaving group authenticate (returns handled) for request 21
Sending Access-Challenge of id 122 to 127.0.0.1 port 32770
        EAP-Message                                                   =
0x01d3006c18013431303030303030303030303030303030303030303030303030303030
303030303030303030303030303030303030313933300233323839333831383631353935
353438393230383532363537333834363135323337363834323432353435303232323136
00        Message-Authenticator = 0x00000000000000000000000000000000
```

```
            State = 0xa1bf1630c1a6b9d028d666d68576dd61
Finished request 21
Going to the next request
--- Walking the entire request list ---
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=123, length=158
            User-Name = "testuser"
            NAS-IP-Address = 127.0.0.1
            NAS-Port = 0
            Message-Authenticator = 0xa431c1dd9bafade1f910fdd94eeea809
            State = 0xa1bf1630c1a6b9d028d666d68576dd61
            EAP-Message                                                  =
0x02d3004e18023336383637363937393032353838333434333139383436343031363332
36313434333034333739383232353138393538300003162 34ba65aa3d0bc50adbc883d1b
ca7c18c207d98b
  Processing the authorize section of RADIUSd.conf
modcall: entering group authorize for request 22
  modcall[authorize]: module "preprocess" returns ok for request 22
  modcall[authorize]: module "chap" returns noop for request 22
  modcall[authorize]: module "mschap" returns noop for request 22
    rlm_realm: No '@' in User-Name = "testuser", looking up realm NULL
    rlm_realm: No such realm "NULL"
  modcall[authorize]: module "suffix" returns noop for request 22
  rlm_eap: EAP packet type response id 211 length 78
  rlm_eap: No EAP Start, assuming it's an on-going EAP conversation
  modcall[authorize]: module "eap" returns updated for request 22
    users: Matched entry testuser at line 80
```

```
   modcall[authorize]: module "files" returns ok for request 22
modcall: leaving group authorize (returns updated) for request 22
   rad_check_password:    Found Auth-Type EAP
auth: type "EAP"
   Processing the authenticate section of RADIUSd.conf
modcall: entering group authenticate for request 22
   rlm_eap: Request found, released from the list
   rlm_eap: EAP/speke
   rlm_eap: processing type speke
rlm_eap_speke: wrong proof
  rlm_eap: Handler failed in EAP/speke
   rlm_eap: Failed in EAP select
   modcall[authenticate]: module "eap" returns invalid for request 22
modcall: leaving group authenticate (returns invalid) for request 22
auth: Failed to validate the user.
Delaying request 22 for 1 seconds
Finished request 22
Going to the next request
Waking up in 6 seconds...
rad_recv: Access-Request packet from host 127.0.0.1:32770, id=123, length=158
Sending Access-Reject of id 123 to 127.0.0.1 port 32770
            EAP-Message = 0x04d30004
            Message-Authenticator = 0x00000000000000000000000000000000
--- Walking the entire request list ---
Waking up in 3 seconds...
--- Walking the entire request list ---
Cleaning up request 21 ID 122 with timestamp 44f51bcf
Cleaning up request 22 ID 123 with timestamp 44f51bcf

Nothing to do.    Sleeping until we see a request.
```