IMPLEMENTATION OF

A NEW 802.1X

PORT AUTHENTICATION METHOD

THESIS

Presented to the Graduate Council

of Texas State University–San Marcos

in Partial Fulfillment

of the Requirements

for the Degree

Master of SCIENCE

by

Yu-Ming Cheng, B.A

San Marcos, Texas

December 2004

DEDICATION

This thesis is dedicated to my lovely family and friends who gave me support and encouragement during this work and to the Computer Science Department at Texas State University–San Marcos.

ACKNOWLEGDEMENTS

I thank many people whose support and encouragement brought me here. Without this help, I would never have finished this work.

First and foremost, I would like to thank my advisor, Dr. McCabe, for his continuous guidance and support. With his invaluable advice and excellent teaching, this two-year graduate thesis research has been one of the most valuable and unforgettable experiences of my life. I truly appreciate his patience and the help he has given me.

Also I am very thankful to the other two members of my thesis committee, Dr. Peng and Dr. Hazlewood, for supporting my thesis work.

Finally, I am extremely grateful to my parents who gave me both life and spiritual support while studying in United States. With their help, I am able to fully concentrate on my studies without worrying about anything. This honor of graduation is dedicated to them.

This manuscript was submitted on November 8, 2004

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

IMPLEMENTATION OF A NEW 802.1X

PORT AUTHENTICATION

METHOD

by

YU-MING CHENG, B.A

Texas State University–San Marcos

December 2004

SUPERVISING PROFESSOR: THOMAS MCCABE

With increasing usage of wireless devices, security issues have become a major challenge for wireless network. In order to solve security defects of the 802.11 standard, the 802.1X authentication standard is adapted to provide additional security. This thesis aims at studying and analyzing 802.1X protocols used to provide security. It includes detailed explanations of various EAP authentication methods which are used by 802.1X, their functionalities and weaknesses, in terms of different security attacks. This paper also describes the advantages and design of an authentication simulator. Then we propose a new EAP authentication method based on the 802.1X protocol and simulate the authentication process based on the simulator.

CHAPTER 1

INTRODUCTION

Security has become a big issue for networks. Because of the Internet and accompanying dependency on computer networks for business environments, preventing attacks or eavesdropping has become a primary task. Some common network security attacks are as follows:

- Eavesdropping

Because network communication occurs in clear text format, attackers can easily listen to the network and monitor all traffic. This is called passive wiretapping. However, sometimes attackers not only listen but also modify or inject something into network called "active" wiretapping, which can cause serious problems. The normal way to prevent active or passive wiretapping is to use data encryption so that attackers can listen on the network but can't decode data nor can they inject modified (unencoded) data.

- Denial of Service

Denial of Service is a situation where a user or organization is deprived of the services of a resource; such as, an attacker floods a network with useless requests so that it can't handle real user requests. By doing this, attackers can either cause a network computer to disconnect from network, crash the computer or otherwise deny access to the service. Denial of Service attack is easy in wireless networks because of the broadcast use of radio signals.

- Dictionary Attack

A dictionary attack involves gathering a list of keys or words in order to break security systems, especially legacy password based systems. Attackers can go through a dictionary beginning with higher probability words and systematically test all probable passwords in systems they attack.

- Session Hijacking

Session Hijacking is a way of stealing a user's identity in an ongoing conversation (session) and impersonating the real user. In wireless environments, attackers can perform hijacking by obtaining packets between two stations and changing an attacker's interface firmware to produce the same MAC address as the real user.

- Man-in-the-Middle Attack

Man-in-the-middle attacks occur when an attacker is able to intercept the communication between two parties. Without knowing there is a man in the middle of the conversation, both parties will assume this communication is secure. The attacker can pretend to be either party by receiving message from one party, modifying it and forwarding it to the other party. This attack becomes fairly easy under wireless environment because everyone is able to intercept radio signals.

A variety of methods have been created to protect against these network attacks, such as using strong data encryption, firewalls, complicated authentication methods such as packet authentication, login authentication, certificate chains and others.

Wireless communication with its portability and flexibility, has become a new trend in communication networks for organizations. However, when compared with wired networks, due to its broadcast nature and communication medium, wireless communication is susceptible to more attacks and requires additional protections. The following are some of the threats and vulnerabilities that a wireless communication network encounters [33]:

- Wireless networks have all the security problems that old conventional wired network have.
- Because of the natural "airwave" character of transmission media, an unauthorized user can access the wireless network, bypassing firewall protections.
- Transmission data without encryption or poor encryption can easily be intercepted and read.
- Anonymous entities may steal others identity and gain access to an internal network with full, authorized privileges.
- Intruders may easily inject bad data to corrupt data on wireless devices and gain access to internal wired network.

In this thesis, we mainly focus on 802.11 wireless local area networks (WLAN) and look at, in general, the security issues associated with the 802.11 standard. Then we will introduce our main subject, the 802.1X port authentication framework and its authentication methods. We will consider advantages and disadvantages provided by 802.1X for 802.11 WLAN and how 802.1X addresses different security issues. We will also look at the need for an authentication simulator, what functionality it requires and how to simulate an authentication process. In the last part of the thesis we will propose a new authentication method based on the 802.1X framework and compare it with different 802.1X authentication methods. The method will be tested with the simulator.

1.2. Organization of the Thesis

The remaining chapters are organized as following:

Section 1.3 addresses current 802.11 standard security issues. We will cover basic 802.11 infrastructure, authentication methods, the concept of using wired equivalent privacy key (WEP) for data encryption and discuss security problems with it.

Chapter 2 discusses the 802.1X port authentication standard and how it provides a security framework on top of 802.11. We will cover details of the EAP and RADIUS protocols

used in 802.1X and the advantage of using these protocols. We also briefly mention new 802.1X authentication methods.

In Chapter 3, we cover 802.1X some authentication methods in detail. We will compare different methods (EAP/MD5, Cisco LEAP, EAP/TLS, EAP/TTLS and EAP/PEAP) and how they handle security problems and attacks, weakness of each method, and methods to counter each weakness.

In Chapter 4, we propose the design and implementation of an 802.1X authentication simulator. We discuss the reasons we need a simulator and how it is used to simulate different authentication methods. We also cover the detailed design of the simulator and the functionality it provides with a sample result of testing an authentication method using the simulator.

In Chapter 5, we propose a new authentication method based on 802.1X and use the authentication simulator to verify the authentication processes with a real, external open source radius server, FREERADIUS. We also look at how our new method counters security attacks compared with current 802.1X authentication methods and present some conclusions.

Chapter 6 presents the latest standard developments using 802.1X. We will look at the WiFi Protect Access (WPA) and new 802.11i standard.

1.3. Current 802.11 Standard Issues

In the IEEE 802.11 standard [1], a WLAN can operate in one of two modes. In infrastructure mode or basic service set (BSS) mode, it is composed of clients, which are computers equipped with wireless network interface cards (NIC), and other computers called access points (AP). Clients communicate to each other through the access points, which also act as bridges between wired and wireless networks. The 802.11 standard defines management frames to provide support for use in infrastructure mode. They are Beacon, which is used by AP to broadcast its existence, Authentication (request and response), Association (request and response), Reassociation (request and response), Dissassociate and Deauthenticate. An ad-hoc mode or independent basic service set (IBSS) network, on the other hand, is only composed of

clients which communicate directly to each other using their wireless NIC, without an intervening an access point. The Common use of ad hoc mode is to create a short-lived network to support a meeting in a conference room. The following figure (source: Corporate Wireless LAN: Know the Risks and Best Practices to Mitigate them) demonstrates differences between two modes:

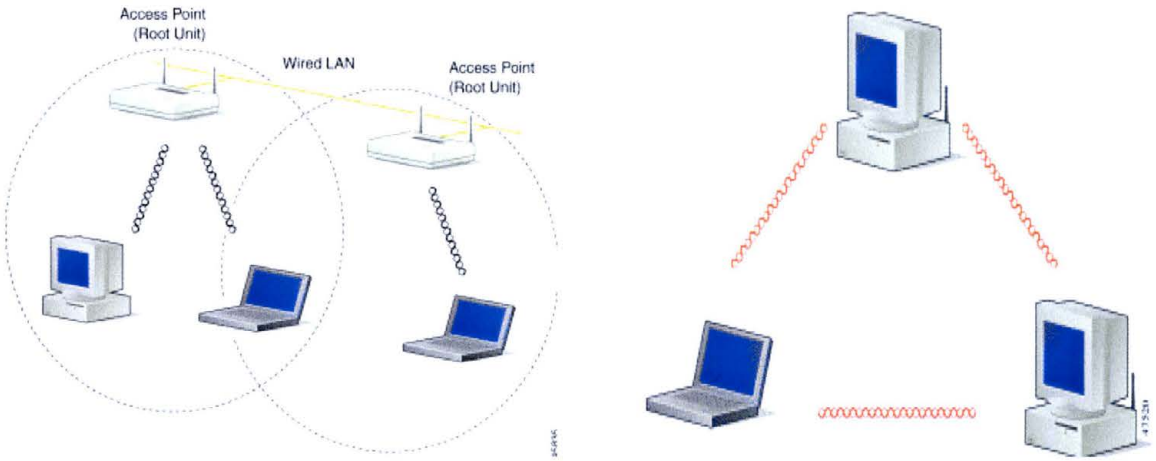

Figure 1-1 802.11 Infrastructure mode (left) and Ad hoc mode (right)

The 802.11 standard contains several security issues. They are described as following:

- Issue 1: Usage of service set identifier(SSID)

The 802.11 standard, for each 802.11 LAN, defines a unique id called the service set id (SSID), which is a network name used to identify an access point in order to limit access. A station has to know the access point's SSID in order to associate with the access point. However, each access point usually broadcasts its SSID in beacon frames; therefore any station can easily associate with an access point based on this SSID. Even though an access point can be configured to not broadcast a beacon, because of the "clear text" form of a SSID in other frames, it is easy for intruders to catch radio signals and find the SSID of a wireless access point.

- Issue 2: Insecure wired equivalent privacy(WEP)

802.11 defines an encryption method called wired equivalent privacy to provide data encryption for preventing eavesdropping. It uses a symmetric algorithm, the RC4 encryption

algorithm. RC4 is a stream cipher which means it takes a short key and expands it into a pseudo-random key stream. The sender can XOR the key stream with data, converting it into cipher text. If an observer catches two cipher texts encrypted with same key, it is possible to begin to discover the short key and hence, to decipher the data stream. WEP address these issues by using an Integrity Check to prevent packet from being modified and an Initialization Vector (IV) to produce different RC4 key streams for each packet. WEP use a 24 bit Initialization and either a 40 or 104 bit WEP key to produce a 64 or 128 bit RC4 key. However, a 24 bit IV was a very poor design decision because it necessitates reuse of the IV after at most 16M transmissions. Then, it will produce the same key stream used before. A busy access point running at 11Mbps could run out of IVs within 5 hours. With this defect, intruders can easily collect two ciphertext packets that are encrypted with same key. The following figure (source: How Secure is Your Wireless Network?: safeguarding your Wi-Fi LAN) demonstrates WEP operation.

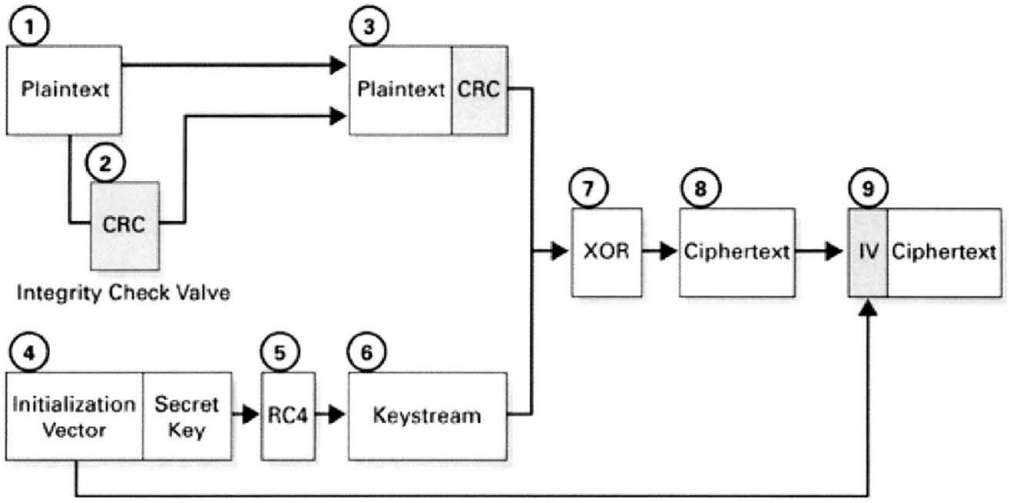

Figure 1-2 WEP encipherment

- Issue 3: Open system and shared key authentication modes.

In the 802.11 standard, a station needs to associate with an access point in order to access its connected networks. The 802.11 standard defines association as a three-state process: (1) unauthenticated and unassociated; (2) authenticated and unassociated; (3) authenticated and associated. Therefore, a station can not get associated with an access point before the

authentication process has completed. Two types of authentication processes are supported by 802.11: open system and shared secret.

For open system, as the name describes, the AP and its network are open to any station. As long as a station knows the access point's SSID, it can send an Authentication-Request. The access point will send an Authentication-Response to indicate success. After the authentication process completes, the client will send an Association-Request to request connection to the AP. Once the AP sends back Association-Response, the client is associated to the AP. In the open method, any anonymous user may get associated with an access point and gain privilege to backend networks. Even though, an open system can use WEP for encryption, it is still a very poor way to prevent security attacks.

For the shared secret authentication method, a station and access point use a predefined WEP secret for authentication. After the station sends an Authentication-Request, the access point will send a random challenge back. The station will use the shared secret as part of a key to hash the challenge and send the hashed value back to the access point. Upon receiving the response, the access point will verify it by hashing and comparing the identical data. The access point will then send an Authentication-Response to indicate success or failure. If the authentication succeeds, the association process can then occur. Although the shared secret method provides better authentication using its encrypted challenge response, since a WEP key is very easy to crack, intruders can easily find the WEP key and produce the same hash value response.

- In conclusion: The 802.11 standard suffers from the following weakness:
  - Weak per packet authentication.
  - Weak user identification and authentication
  - No central authentication support
  - No dynamic key generation and management.

CHAPTER 2

802.1X PORT AUTHENTICATION FRAMEWORK

2.1. 802.1X Port Authentication Protocol

The 802.1X standard [2] was originally defined as port based access control protocols for authenticating and authorizing devices that try to attach to a LAN port. The idea was to prevent anyone from connecting to a network by plugging an Ethernet cable into a hole in the wall without checking the user's identity and authorization status or to prevent unauthorized connections via modems over the public telephone system. The 802.1X protocols provide advantages such as user identification, centralized authentication control and key management. Due to the security weaknesses of 802.11 standards previously mentioned, 802.1X has been adapted for wireless LANs to overcome some security problems inherent in 802.11 by offering an overlay of new authentication methods. The architectural relationship between the 802.1X and 802.11 standards is indicated in the following figure (source: www.cisco.com):

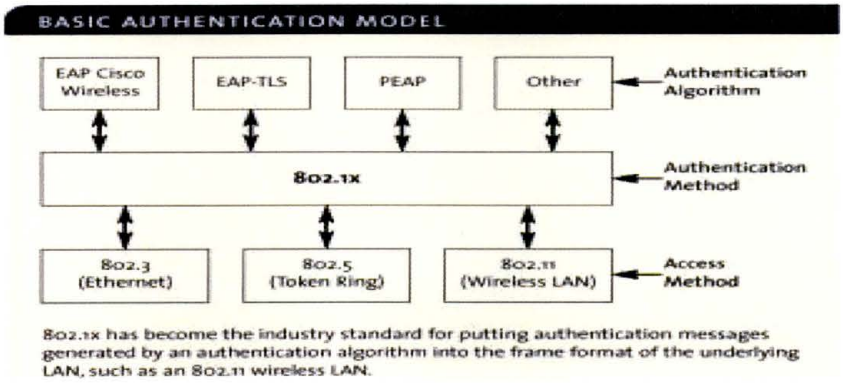

Figure 2-1 802.11 and 802.1X architecture relationship

As the above figure shows, 802.1X provides extra authentication methods on top of any 802 network. It uses a frontend authentication algorithm, Extensible Authentication Protocol (EAP) with a backend centralized authentication control protocol, such as Remote Authentication Dial in User Service Protocol (RADIUS), to implement the entire authentication process. In order to perform its port control authentication mechanism, it defines three important actors: Client/Supplicant, Authenticator, and Authentication Server

Authenticator – An Authenticator is an entity which enforces authentication before allowing external devices to attach to its ports in order to access most LAN services. It is responsible for determining an authorized port's state according to the response from an authentication server. In wireless LANs, the authenticator is usually an Access Point.

Supplicant – A supplicant is an entity which desires access to the ports offered by authenticator (AP). It can either initiate authentication by supplying its credentials or respond to identity requests initiated by an authenticator. In wireless LANs, a supplicant can be any device with a wireless adapter wishing to access an AP.

Authentication Server – An authentication server performs the authentication process, validates a supplicant's credentials, then indicates whether the supplicant is authorized to access the authenticator's port services or not. The Authentication server is typically a RADIUS server accessed over EAP transport (see discussion below).

The following figure (source: Introduction to 802.1X for Wireless Local Area Networks) demonstrates the 802.1X architecture and the relations between the three different actors.

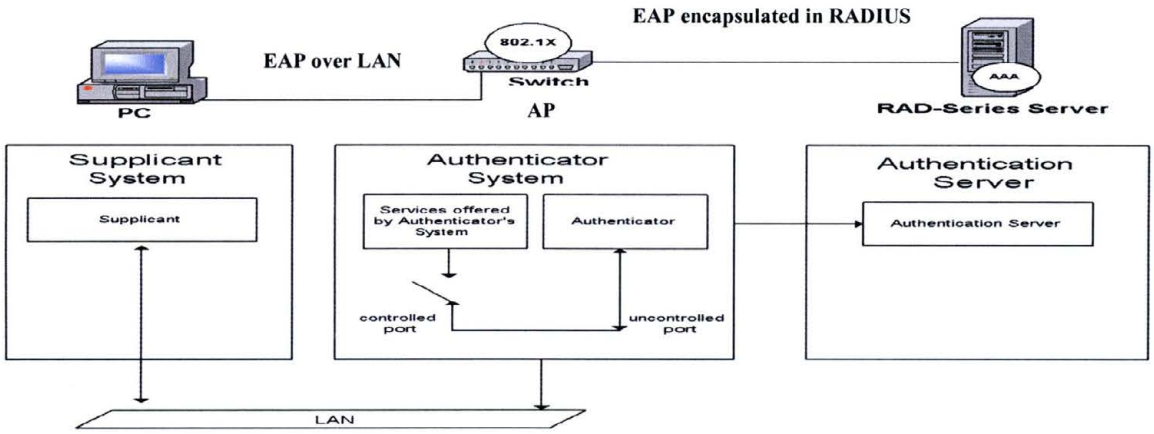

Figure 2-2 802.1X architecture overview

When a new supplicant attaches to a port, which is a logical port in wireless environments, an authenticator will set its controlled port into Unauthorized state and block all messages through the port with the exception of special EAP messages. The authenticator will forward received EAP messages to the appropriate authentication server, where the authentication check takes place. The server usually has access to a global database for authentication certification. Upon receiving a response from the authentication server, an authenticator will set a port into Authorized state if authentication succeeds or set it to the Unauthorized state and the supplicant fails authentication. Once authentication succeeds, the supplicant can access the port and the attached wired and wireless LANs.

2.2. Extensible Authentication Protocol

EAP was defined in RFC 2248 [18], (made obsolete by RFC 3748 [07]), and was originally created as an extension to the Point to Point protocol (PPP) [23] to provide additional authentication support. It usually runs over a data link without requiring IP addresses; therefore, any protocol that carries frames can be used for EAP. A general EAP frame format is defined as shown in the following figure (source: Examining 802.1X and EAP):

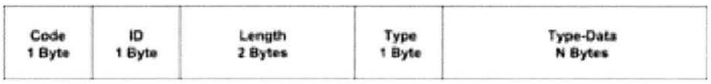

Figure 2-3 EAP frame format

The Code field indicates the type of EAP packet. EAP defines four types of packets to be carried. They are EAP-REQUEST (code = 1) which carries a request from authenticator to supplicant; EAP-RESPONSE (code =2) which allows a supplicant to send a response to an authenticator; EAP-SUCCESS (code = 3) which indicates success of authentication; and EAP-FAILURE (code = 4) which indicates failure of authentication. For every request and response packet, there is a type field which indicates the type of EAP authentication method to be used, such as Identity (type = 1), Nak (type = 3) and MD5-Challenge (type =4). In order to provide extra authentication for current 802.11 broadcast networks, the 802.1X framework uses the EAP protocol to carry

protocol messages over wired or wireless LANs, which extends EAP from EAP over PPP to EAP over LAN (EAPOL) or EAP over Wireless (EAPOW). 802.1X defines four new message formats for EAPOL: EAPOL-START, which is used by a supplicant to signal an authenticator to start the authentication process; EAPOL-KEY, which is used by an authenticator to send encryption keys to a supplicant, once the authentication process succeeds; EAPOL-PACKET, which is used to carry the actual EAP message over LAN; and EAPOL-LOGOFF, used by a supplicant to inform an authenticator that it is disconnecting from the network port. Only one of the EAPOL message types is used to carry traditional EAP messages, the rest provide administrative support. The advantages of using EAP include:

- EAP supports multiple authentication mechanisms without having to pre-negotiate a specific one.
- An authenticator may act as a "pass through" and just forward a message to an authentication server; therefore it doesn't have to understand multiple EAP authentication methods. Only a server has to decide which authentication protocols are supported and negotiate each with a supplicant.
- By separating the roles of authenticator and authentication server, an authenticator only enforces authentication status at the port. Credential management is simplified, being isolated to the authentication server.

However, in order to carry out the entire EAP process in 802.1X, all three entities have to support the EAP protocol. But by separating the authenticator and authentication server, an extra authentication protocol (RADIUS) has to be used in order to carry EAP messages. This increases the complexity of security issues and key distribution [25].

EAP itself is not an authentication protocol. It's only a standard that describes how a message is carried and exchanged between supplicant and authentication server. It relies on other protocols for authentication. The commonly used methods include: EAP-MD5, Cisco LEAP, EAP-TLS, EAP-TTLS and PEAP. By choosing the proper EAP authentication method, 802.1X authentication can provide a counter to eavesdropping, dictionary attacks, session hijacking and man-in-the-middle attacks.

2.3. Remote Authentication Dial in User Service Protocol

RADIUS is an authentication, authorization, accounting (AAA) [10] protocol that is used to provide network access control. It is a connectionless, UDP-based protocol and thus, it requires an underlying IP network. RADIUS authentication was defined in [11]. The RADIUS packet format is composed as following figure (source: http://ing.ctit.utwente.nl/WU5/D5.1/Technology/radius/):

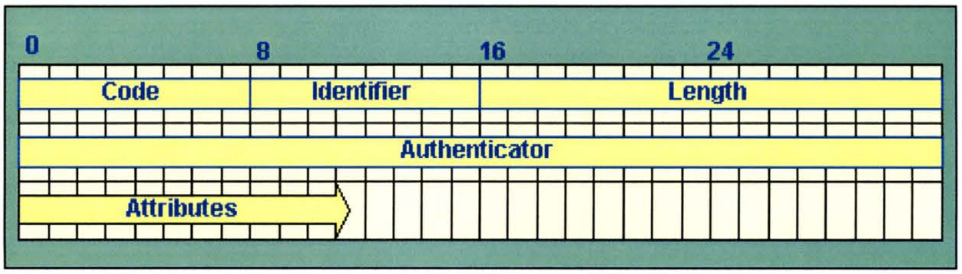

Figure 2-4 RADIUS Frame Format

The Code field defines the type of RADIUS packet. It contains four types of packets for authentication. They are ACCESS-REQUEST (code = 1), which carries request from authenticator to authentication server; ACCESS-CHALLENGE (code = 11), which used by an authentication server to issue challenge to an authenticator; ACCESS-ACEPT (code = 3), which indicates success of the authentication; and ACCESS-REJECT (code = 4), which indicates failure of the authentication. Both authentication server and authenticator authenticate each packet by using a predefined shared secret. In order to prevent a packet being modified or prevent replay attacks, every ACCESS-REQUEST packet contains a 16 octet randomly generated Request-Authenticator field. When an authentication server sends back a response, it contains a Response-Authenticator field which is a MD5 hash value of code, identifier, length, Request-Authenticator, attributes and shared secret. Upon receiving this challenge, the authenticator is able to check if a packet has been modified or not.

RADIUS also defines several attribute fields in order to carry different types of data. In order to support EAP in RADIUS, two types of attributes are used: EAP-MESSAGE and MESSAGE-AUTHENTICATOR [08]. The EAP-MESSAGE attribute allows an EAP message to be encapsulated within RADIUS, which is referred to as EAP over RADIUS and the EAP message is passed from authenticator to server. The MESSAGE-AUTHENTICATOR attribute is present only

when an EAP message is encapsulated inside RADIUS. It provides ACCESS-REQUEST packet integrity protection by calculating the MD5 hash of the whole packet. Although RADIUS is not part of 802.1X, it has become a de facto back-end protocol used in 802.1X to carry EAP message between authenticator and authentication server [19].

Even though RADIUS provides a method to support authentication, it has security weaknesses [29].

- Shared secret based attack

Because Response-Authenticator field and MESSAGE-AUTHENTICATOR attribute are based on the MD5 hash of a shared secret, by capturing Response-Authenticator or MESSAGE-AUTHENTICATOR, the RADIUS shared secret is vulnerable to offline dictionary attack by pre-computing the MD5 hash of the packet and then resuming the hash for each guess. In addition, authenticator and server only authenticate each other through the shared secret. RADIUS allows use of the same shared secret between multiple authenticators. This provides more data for attackers to use and any compromised client can compromise other machines. Most implementations of RADIUS allow the shared secret to be an ASCII string with no more than 16 characters. This implementation limit makes it easier to guess the shared secret.

- Request-Authenticator based attack

The whole packet protection mechanism in RADIUS is based on the authenticator field, which means its protection depends on how the random and unique Request-Authenticator field is generated. Therefore, if these random numbers collide, attackers can use collisions to masquerade as the server and replay responses. Also attackers can predict and create server responses to masquerade as valid server response and produce seemingly valid ACCESS-REJECT packets, creating a denial of service attack.

2.4. 802.1X Authentication Process

As mentioned in section 2.1, 802.1X provides an authentication framework on top of 802.11. Therefore, the 802.1X authentication process starts after the 802.11 Association-Response frame has been received from the supplicant. The authentication process can be divided into two parts: EAP conversation between supplicant and authenticator and RADIUS conversation between authenticator and server. The process starts by a supplicant signaling the authenticator with a EAPOL-START packet which indicates the starting of the EAP process. After receiving the start signal, an authenticator will issue an EAP-REQUEST with type field set to Identity to request the supplicant provide proper identification for the server. Once the authenticator receives EAP-RESPONSE, it will encapsulate the response in a RADIUS EAP-MESSAGE attribute and send a RADIUS ACCESS-REQUEST packet to an authentication server. After the server receives the packet, if the user identity matches the server's global database, it will start negotiating with the supplicant by sending an EAP message with type field indicating which EAP authentication method the server wishes to use. If supplicant agrees with the authentication method, the choosen authentication process will continue back and forth from supplicant to server until the server issues either ACCESS-ACCEPT for success or ACCESS-REJECT for failure. Then the authenticator will forward the outcome of the server message by issuing EAP-SUCCESS or EAP-FAILURE to the supplicant. The following figure (source: Introduction to 802.1X for Wireless Local Area Networks) describes the whole 802.1X process with RADIUS authentication:

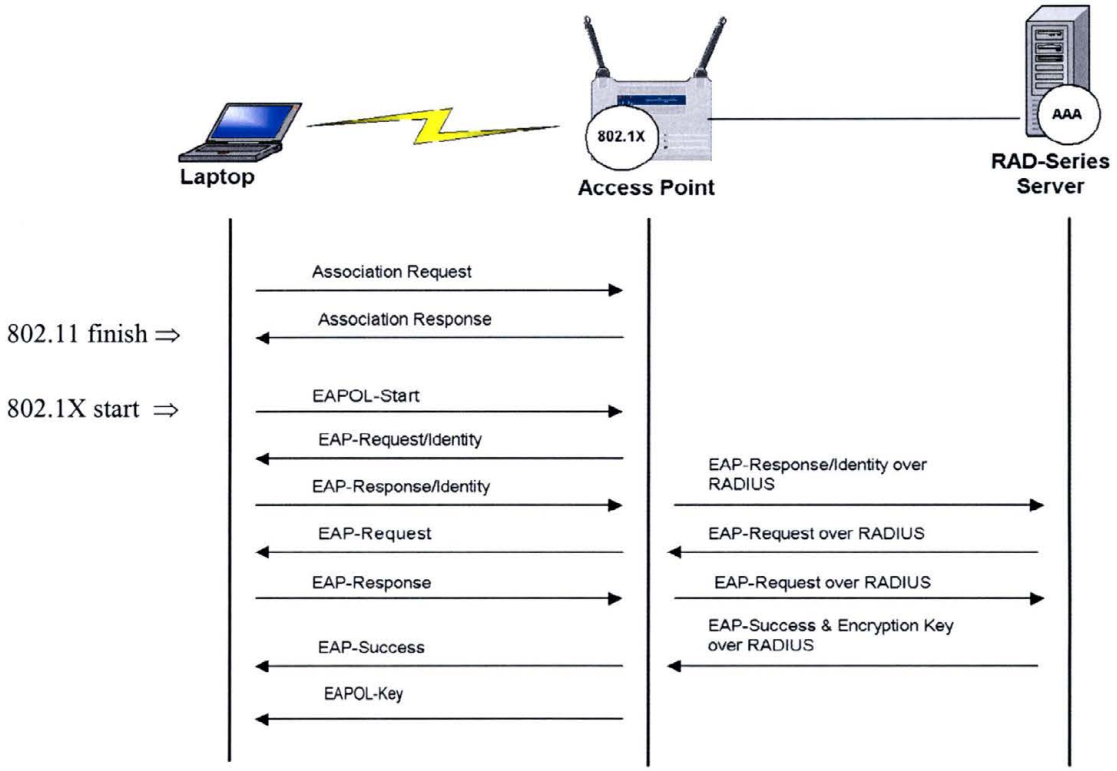

Figure 2-5 802.1X authentication process

The last two messages of the figure above, which are Encryption Key passed from server to authenticator and then, EAPOL-Key sent from authenticator to supplicant, will only occur if the chosen EAP method supports key generation. Although 802.1X does define the passing of keys, it doesn't require an underlying EAP method to support key material and key generation. The use of EAPOL-Key is for the authenticator to transmit an encrypted key to the client. It includes the transmission of unicast key, which is only used between authenticator and client for data encryption, and a broadcast key used between AP and a group of clients that associate with it.

CHAPTER 3

802.1X EAP AUTHENTICATION METHODS AND KEY MANAGEMENT

3.1. Password Based EAP Authentication Methods

Among authentication methods currently supported for EAP authentication, common methods that have been implemented can be divided into two categories: password based authentication and certificate based authentication. In password based authentication, the client and server authenticate each other through a predefined password. The password based EAP authentication methods include:

- **EAP-MD5**

EAP-Message-Digest-5 is the basic authentication method defined in the EAP standard. It is based on a MD5 hash of the supplicant's username and password. The authentication process starts after a server receives a client's IDENTITY-RESPONSE. If the server is configured to use EAP-MD5, it will issue an EAP message that contains a 16 octet randomly generated challenge with type field set to MD5-Challenge, encapsulated in a RADIUS ACCESS-CHALLENGE packet. Once the client receives this EAP message, if it supports EAP-MD5, it will calculate the MD5 hash of the concatenation of packet id, user password and the challenge and generate a 16 octet response to send back to the server. Since the server knows all the elements of the MD5 hash, it can also compute the MD5 hash value and compare it with the received response. If these two values match, the authentication process succeeds, otherwise it fails.

EAP-MD5 provides a very lightweight authentication and is very easy to deploy; however it has serious security problems when used in a wireless environment. First, it only provides one way authentication (server authenticates client), which makes it susceptible to a man-in-the-middle attack. Second, it doesn't generate any keys that can be used for subsequent channel encryption. Therefore, the supplicant and AP have to use WEP keys for data encryption. Since WEP can be attacked, so can EAP-MD5. Because EAP-MD5 is a password based authentication, it may suffer from dictionary attacks. Although 16 octet challenges provide less possibility for attack, attackers with large dictionaries may still break through.

- **Cisco LEAP**

LEAP [26], which stands for Lightweight EAP, is not part of the EAP standard. It was developed by Cisco and used primary for Cisco wireless environments. It is also a password based authentication. Unlike EAP-MD5, LEAP does provide mutual authentication and key generation for data encryption. The LEAP process is similar to EAP-MD5 except there are two challenges: one for the client and another for the server. LEAP uses 8 octet challenges with 24 octet responses computed by using Microsoft's MS-CHAP [16] algorithm. The 24 octet client response is calculated by a function NTChallengeResponse() defined in [16], using the server's challenge and Unicode of the password as input. On the other hand, the 24 octet server response is generated by calculating 16 octets of MPPEHASH then using the client's challenge and MPPEHASH as input to function ChallengeResponse(). After the server verifies client authentication, it starts the client-authenticates-server process. Depending on the result of the server authentication, the server then issues EAP-SUCCESS or EAP-FAILURE. Finally, by the time mutual authentication succeeds, server and client will each have independently computed the same intermediate key by using the Microsoft Point to Point Encryption (MPPE) [13] function and then have generated a 34 octet session key. Since authentication occurs only between client and server, the authenticator has no way of knowing how to generate the key. Therefore, the server will now send the computed key in an ACCESS-ACCEPT packet to the authenticator.

Upon learning the session key, the authenticator and client can further negotiate keys for data encryption.

LEAP prevents man-in-the-middle attack by using mutual authentication. Also by generating a session key and distributing it to the authenticator, LEAP is able to generate a dynamic WEP key between client and authenticator for strong data encryption. Unfortunately, LEAP has already been broken. A researcher, Joshua Wright, was able to go through large dictionary files very quickly and mount an offline dictionary attack against LEAP [30].

3.2. Certificate Based EAP Authentication Methods

All certificate based EAP authentication methods have two important components. They are described as following:

3.2.1. Certificate and Certificate Authority

In public key infrastructure (PKI) [9], two keys are used for encryption. One is a private key and the other is a public key. As the name suggests, the public key is usually published and known to anyone. However, when two parties try to communicate securely by using both sites' public key for encryption, unless they are "directly" told the value of the public keys, it is very difficult to distinguish real public keys from fakes. The issue of certifying whether a public key really belongs to the "person" you are expecting to talk to has become more important with the increasing use of e-commerce. Certificates were created to provide a secure way of publishing verifiable public keys. When a trusted third party, a certificate authority (CA), creates a certificate for a user's public key, the public key can be more trusted. If an entity is presented with a public key within a certificate, it can request the CA to verify the contents of the certificate, using the CA's digital signature system. In other words, if you can trust the CA then the assumption is that you can trust the contents of any certificate it has signed, including the one just presented to you. Certificates are at the heart of modern authentications where the two parties are unknown to each other before the interaction begins.

3.2.2. Transport Layer Security Protocol

Transport Layer Security (TLS) Protocol is the standard version of Secure Sockets Layer (SSL) Protocol. SSL was developed originally by Netscape for providing secure transactions in web browser/server interactions and is based on the use of digital certificates. After SSL was standardized by the IETF, it was renamed TLS, defined in [21]. The difference between TLS and SSL is that TLS does not concern itself with browsers but is entirely concerned with the transport protocol layer. TLS consists of two internal layers and several sub-protocols. The TLS layer structure can be described as the following figure indicates (source: Internet Application Security):

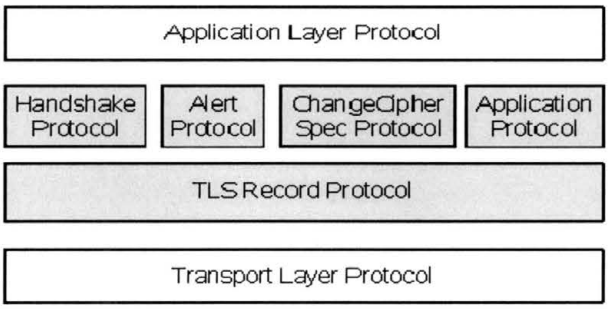

Figure 3-1 TLS layer structure

TLS Record Protocol is used to provide a private and reliable connection; it is responsible for transporting encrypted data between two peers using the encryption parameters agreed to in the handshake protocol. Handshake Protocol provides the connection with a reliable and secure negotiation of encryption and cryptographic keys to be used in the connection. It also performs peer identity authentication.

A relation between two parties is established in TLS by using a complicated handshake algorithm. At the end of the handshake, both parties have computed a TLS master key and have agreed to a secure channel where only the two end peers can "read" the contents of the channel. This channel is often referred to as a secure tunnel through the intermediate network. Starting from this point, all data are encrypted using a master key. The TLS handshake exchange is presented in the following figure (source: http://www.cisco.com/warp/public/759/ipj_1-1/images/fig2SSL.gif):

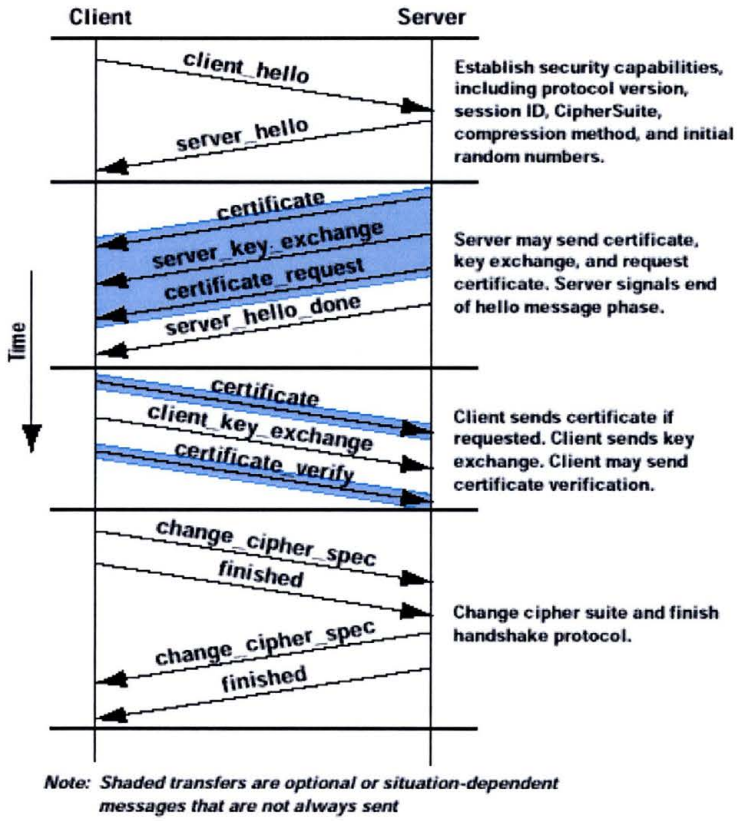

Figure 3-2 TLS handshake process

3.2.3. Common Certificate Based EAP Methods

- **EAP-TLS**

EAP-TLS was defined in [5], which defines how TLS can be directly transported over EAP instead of using TCP/IP. The idea is that TLS packets are encapsulated inside EAP messages. EAP-TLS requires both client and server authenticate each other by using digital certificates in the TLS handshake.

The EAP-TLS authentication process starts after the server receives the client's IDENTITY-RESPONSE. The server will issue an EAP-REQUEST with TLS start flag set. Once the client receives this start flag messages, the TLS handshake exchange starts with the TLS handshake encapsulated and carried inside EAP. After both parties finish the handshake, the

server then derives the AAA-key and passes it to the authenticator using Microsoft Vendor-specific RADIUS Attributes [14] inside an ACCESS-ACCEPT packet with EAP-SUCCESS. By passing an AAA-key in an ACCESS-ACCEPT packet to the authenticator, the authenticator is able to synchronize and compute a Transient-Session-Key identical to the client's for future encryption. Since the authenticator is acting like a pass-through, it doesn't need to understand TLS to carry out the process. It only decides the port's authorized status according to EAP-SUCCESS or EAP-FAILURE delivered from the server at the end of the process.

EAP-TLS provides security protections which include mutual authentication by certificate and dynamic WEP key generation. EAP-TLS is difficult to deploy because of certificate management. It also inherits all security issues that affect the TLS protocol.

- **EAP-TTLS**

EAP-TTLS is a short name for EAP Tunneled TLS. EAP-TTLS was defined in an IETF draft [20]. It is an extension method to EAP-TLS and provides strong encryption without the complexity of using mutual certificates on both client and server. The idea is to use TLS to establish a security tunnel first and then use more authentication methods inside the tunnel. The inner authentication methods can be non EAP protocols such as PAP, CHAP [24], MS-CHAP and MS-CHAP-V2 [15] or EAP methods such as EAP-MD5.

The authentication process of EAP-TTLS can be divided into two phases. Phase one is the same as EAP-TLS except it only requires the server to be authenticated to the client. Therefore, the server CERTIFICATE-REQUEST and client's CERTIFICATE handshake will not appear in the TLS handshake exchange. Also after the FINISH handshake has been sent, the server will not issue an ACCESS-ACCEPT with EAP-SUCCESS. Both parties establish a secure tunnel by using a derived Transient-EAP-Key for data encryption between server and client. Once the secure tunnel between client and server has been established, the authentication process will enter phase two. One advantage of EAP-TTLS is it provides identity protection by giving an option for the user to provide its real identity inside the secure tunnel. Therefore, in phase one,

even if the server requests the user to provide its identity, a user can use an anonymous identity instead, and provide its real identity in phase two's identity request.

Phase two will start by the server sending EAP-REQUEST with type field Identity encapsulated inside an ACCESS-REQUEST packet. If the client did not provide its real identity in phase one, it will give its real identity in EAP-RESPONSE. Upon receiving the client's real identity, the server will start the second part of the authentication process, which depends on the inner method chosen. If the inner method's authentication process succeeds, the serve will issue an ACCESS-ACCEPT to the authenticator and indicate success.

Since EAP-TTLS is an extension of EAP-TLS, it inherits all security protections that TLS has, such as mutual authentication, dynamic WEP key generation etc. It also provides extra user identity protection. However, one recent research paper [32] has pointed out that EAP-TTLS can suffer from a man-in-the-middle attack. The main reason is that the inner authentication method is not aware of the existence of the secure tunnel. An attacker can first form a secure tunnel to the server without the server validating the client's identity (only server is authenticated to client but not vice versa). Then the attacker can pretend to be an authenticator asking a client for authentication information, passing it to the server through the secure tunnel. In other words, the man-in-the-middle attacker can authenticate to the server by getting the appropriate answers from an actual, unsuspecting client. Once the inner authentication process succeeds, the attacker can just deny the real client and gain access to the port.

- **PEAP**

PEAP is the acronym for Protected EAP, which was defined in an IETF's draft [4]. PEAP actually is a special case of EAP-TTLS by allowing only EAP authentication methods as an inner method. PEAP authentication procedure is the same as EAP-TLS. Phase one is used to establish a secure tunnel and phase two runs an EAP method inside the tunnel. Microsoft implements PEAP by choosing EAP-MS-CHAP-V2 [22] as its inner method. Besides mutual authentication, dynamic WEP key generation and identity protection, PEAP also provides extra protection against man-in-the-middle attacks by using cryptographic binding of the keys. In normal EAP-

TTLS, after authentication success, client and server generate AAA-key using TLS keying material. When performing a man-in-the-middle attack, only the attacker and server form a secure tunnel. Therefore, attackers can generate AAA-key once the authentication process is complete. PEAP solves this problem by combining TLS and the inner method keying materials together and generating an AAA-key using both. In this situation, an attacker no longer can compute the correct AAA-key since he only knows TLS keying material but not the inner method keying material. In order to use cryptographic binding [17] to protect against man-in-the-middle attacks, any EAP method that doesn't support generation of keying materials can not be used as an inner method for PEAP.

The following table (source: A Technical Comparison of TTLS and PEAP) lists detailed comparisons among three certificate based authentication methods.

	EAP-TLS (RFC 2716)	**TTLS (Internet draft)**	**PEAP(Internet draft)**
Protocol Operations			
Basic protocol structure	Establish TLS session and validate certificates on both client and server	Two phases: (1) Establish TLS between client and TTLS server (2) Exchange attribute-value pairs between client and server	Two parts: (1) Establish TLS between client and PEAP server (2) Run EAP exchange over TLS tunnel
Inner methods	None	Any(Non EAP or EAP)	Any EAP method
Fast session reconnect	No	Yes	Yes
WEP Integration	Server can supply WEP key with external protocol (e.g. RADIUS extension)		
PKI and Certificate Processing			
Server Certificate	Required	Required	Required
Client Certificate	Required	Optional	Optional
Cert Verification	Through certificate chain or OCSP TLS extension (current Internet draft)		
Client and User Authentication			
Authentication direction	Mutual: Uses digital certificates both ways	Mutual: Certificate for server authentication, and tunneled method for client	Mutual: Certificate for server, and protected EAP method for client
Protection of user identity exchange	No	Yes; protected by TLS	Yes; protected by TLS

Table 3-1 Comparison of Certificate Based EAP Authentication Methods

3.3. EAP Key Management

Although 802.1X defines a new message type, EAPOL-KEY to pass keying material between authenticator and supplicant, it doesn't define how the key is generated or how it is used. All key generations depend on which EAP authentication method is chosen. For example, EAP-TLS relies on keying material generated from the outer TLS protocol. In order to provide overall details about key generation and management, an IETF internet draft [6], EAP Key Management Framework, provides a framework for generation, transport and usage of keying material generated by EAP authentication methods. In this draft, if an EAP method supports key derivation, it defines three phases of the protocol as the following figure indicates (source: EAP key management framework):

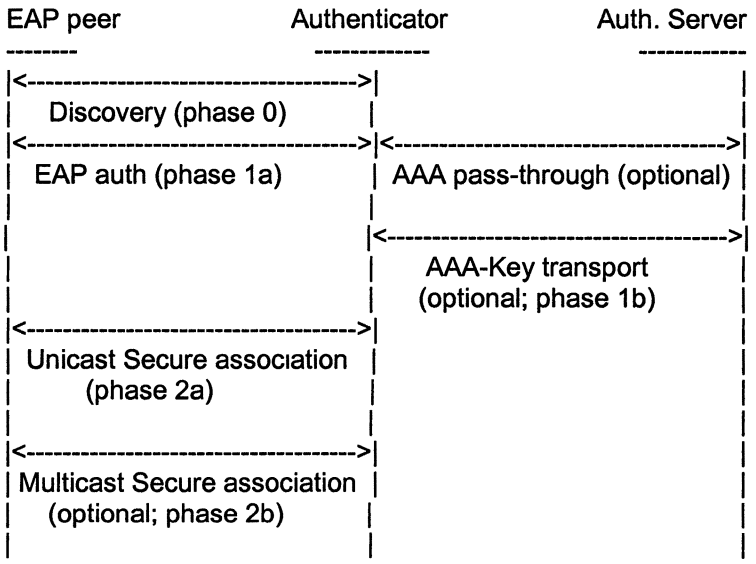

Figure 3-3 Conversation Overview

Phase 0 is just a simple process for a client to discover the location of the authenticator. For wireless environment, the discover phase occurs when a client computer tries to discover and associate with an AP's SSID. In phase 1a, EAP authentication takes place. After phase 1a succeeds, phase 1b will occur only if the EAP method supports key generation. Server and client will then both compute an AAA-key according to keying material derived from the EAP method and the server will pass the AAA-key to the authenticator. Once client and authenticator

both have an AAA-key, phase 2 secure association starts between client and authenticator to exchange information, and then mix it with the AAA-key to create unicast (phase 2a) and broadcast (phase2b) secure channels. Exchanging information during the secure association phase guarantees the mutual proof of possession of the AAA-key, which demonstrates both client and authenticator have been authenticated to each other and authorized by the backend server. This phase also generates a fresh Transient-Session-Key using exchanged information and the AAA-key. Since the Transient-Session-Key is not directly derived from AAA-key, this protects against compromise of AAA-key and assures freshness of Transient-Session-Keys.

CHAPTER 4

802.1X AUTHENTICATION SIMULATOR

4.1. Necessity of Simulator

In order to establish an infrastructure for a 802.1X authentication framework, the following concerns have to be satisfied:

- Three entities; a supplicant which is a computer with wireless NIC card; an authenticator which is an access point; and an authentication server which is a RADIUS server on a wired network have to be present.
- Although an authenticator acts as a pass through, it has to have both EAP and RADIUS protocols implemented in it.
- Both client and authentication server need to support the same EAP authentication methods in order to complete any authentication process.

Currently many 802.1X capable APs, clients and RADIUS servers have been implemented by different universities, organizations and companies. The following table lists some of the available 802.1X supplicant and authentication servers with the supported EAP authentications.

	EAP-MD5	EAP-TLS	EAP-TTLS	PEAP	LEAP
RADIUS Server Support	Cisco, FreeRADIUS, Funk, Interlink, Meetinghouse, Microsoft, Radiator	Cisco, FreeRADIUS, Funk, Interlink, Meetinghouse, Microsoft, Radiator	Funk, Interlink, Meetinghouse, Radiator	Cisco, Funk, Interlink, Meetinghouse, Microsoft, Radiator	Cisco, FreeRADIUS, Funk, Interlink, Meetinghouse, Radiator
Supplicant Client Support	Funk, Open1X, Meetinghouse, Microsoft	Cisco, Funk, Meetinghouse, Microsoft, Open1X	Alfa-Aniss, Funk, Meetinghouse, Open1X	Funk, Meetinghouse, Microsoft	Cisco, Funk, Meetinghouse

Table 4-1 Available EAP support for Supplicant and Authentication Server

As the above table indicates, most only support common EAP authentication methods as described in Chapter 3. New EAP authentication methods have been proposed, but only a few of them are implemented or supported. In this thesis project, we developed an 802.1X authentication simulator to provide a first hand and easy implementation framework for EAP authentication development without using actual wireless devices. Development of an AP and client simulator was necessary because of the complexity and cost of implementing new security protocols in real devices. Normally the protocols used by a wireless supplicant are actually implemented in firmware delivered with the device. Modifying such firmware would require both tedious and tricky reverse engineering. The same would be true in modifying an AP.

The simulator developed in this project includes both supplicant and authenticator actors and can simulate EAP authentication methods, and the simulator can interact with a real, external RADIUS server. Also, by testing newly implemented methods with the simulator, it is possible to test and analyze security issues that an authentication method might suffer.

4.2. Simulation Work

The authentication simulator developed in this thesis project implements the EAP "conversation" by using message queues (EAP over Message Queues) between the authenticator and supplicant. But the communication between authenticator and server uses UDP over an IP network, with both entities either on the same machine or on separate machines. Both authenticator and supplicant simulators are composed of their own state machines (authenticator has five state machines and supplicant has three). With the control and interaction between state machines, these two simulators are able to instantiate the EAP conversation and pass an EAP message encapsulated inside a RADIUS packet to the server. The simulation process is described by the following figure:

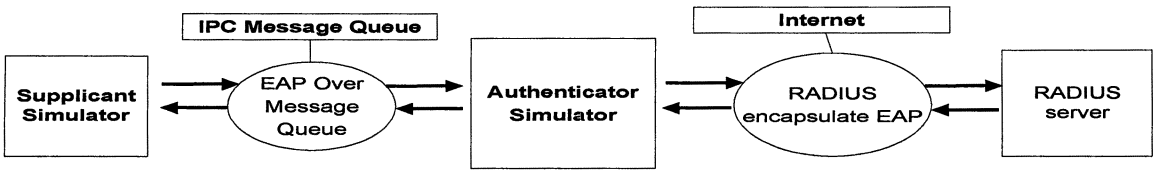

Figure 4-1 Simulation Process

The steps of a single authentication simulation are as follows:

1. For each port, the authenticator creates a message queue available to any supplicant which wants to connect, and then blocks for a port connection request.

2. The supplicant will run its three concurrent state machines which signal the authenticator for a port connection with EAPOL-START, through the message queue.

3. After the authenticator receives EAPOL-START, it will fork a child process to handle the supplicant; the child process runs the five state machines concurrently. According to the state and port status controlled by the state machine, the authenticator child will issue an EAP-REQUEST with type field set to Identity to supplicant, through the message queue.

4. Once the supplicant receives the requests, its state machine will handle this request by sending EAP-RESPONSE including its identity to authenticator, through the message queue.

5. Upon receiving response, the authenticator's state machine will encapsulate EAP messages inside a RADIUS ACCESS-REQUEST packet, and send it to the server over UDP.

6. When the server issues an ACCESS-CHALLEGE to the authenticator, the authenticator will respond by decapsulating the EAP message and then send it as EAP-REQUEST to the supplicant, via the message queue.

7. The simulation process will keep going back and forth until the server issues ACCESS-ACCEPT or ACCESS-FAILURE. Depending on the result from the server, the authenticator's state machine will change to the corresponding state (Authenticated or Failure) and set the port status to either Authorized or Unauthorized.

4.3. Software Design Architecture

A dependency graph for the software modules appears as follows:

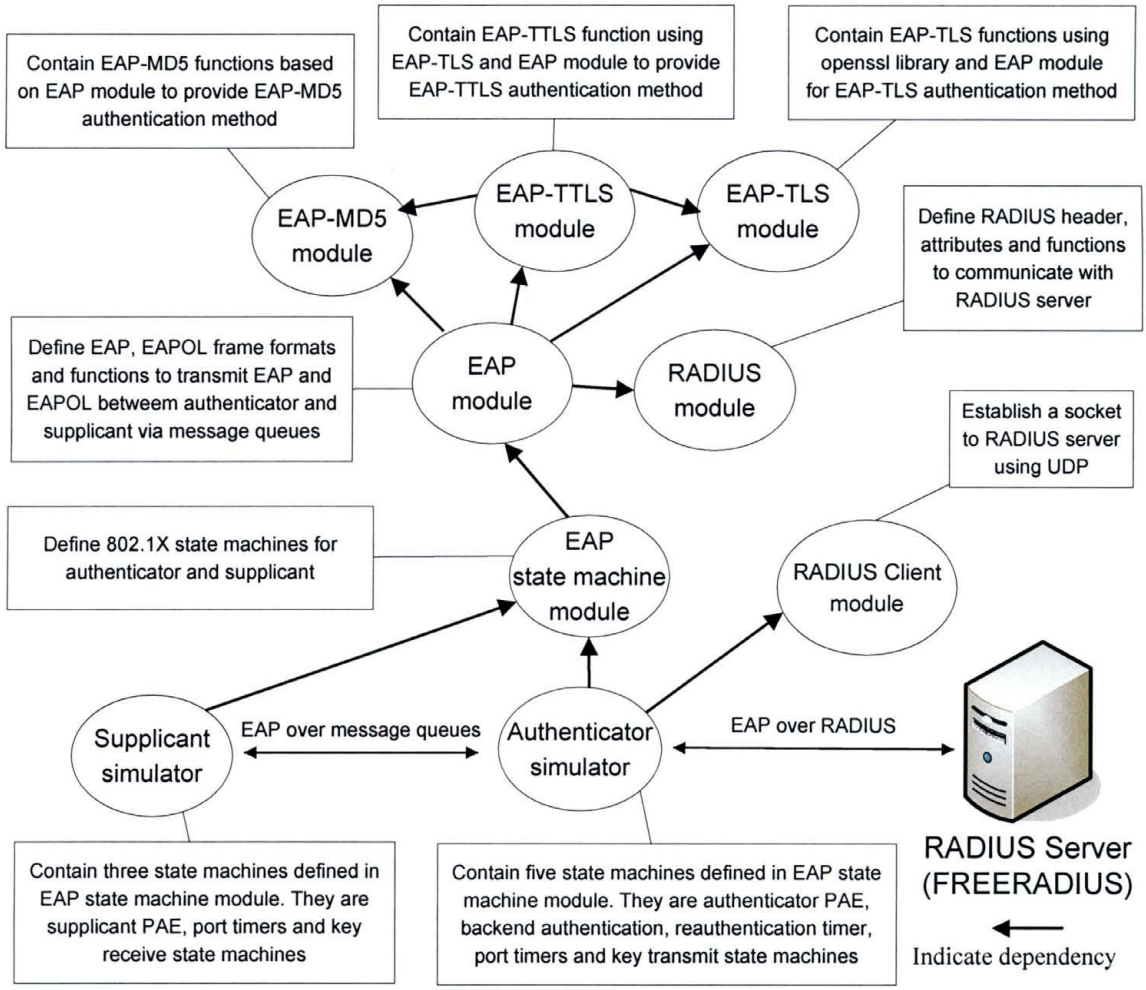

Figure 4-2 Module Dependency Design Diagram of Simulator

The implementation of the authentication simulator consists of two parts: Authenticator and Supplicant which are based on open source implementations of 802.1X authenticator, HostAP and supplicant, WPA_Supplicant. Instead of carrying EAP over wireless between supplicant and authenticator, the simulator uses System V interprocess communication (IPC) message queues, to pass EAP messages (EAP over message queues) which simplifies the wireless signal and hardware requirements. After the authenticator receives EAP messages, it communicates with real, external RADIUS server using RADIUS protocol via LANs. It can accept a maximum of four concurrent supplicant connections. The software architecture of the simulator

relies on different external modules and two main simulator programs, authenticator and supplicant. They are described as follows:

- **Modules**

The EAP module defines EAP and EAPOL frame formats. It also implements different EAP functions which can be used by the authenticator and supplicant to carry on an authentication process. These functions are called by EAP state machines. Depending on the EAP authentication method chosen from server, the EAP functions will use a function implemented in the EAP authentication method module to perform authentication.

Three EAP authentication method modules are responsible for providing functionality to perform different authentications. In the EAP-MD5 module, one function is used to build EAP-MD5 responses. The EAP-TLS module defines TSL flags and SSL structures to perform the TLS protocol by using the openssl library. Because EAP-TTLS is based on EAP-TLS, the EAP-TTLS module uses the EAP-TLS module to establish a secure channel (phase one) and then uses the EAP-MD5 module to perform the inner authentication method (phase two).

The RADIUS module defines RADIUS frame formats and different attributes. It also provides functions for an authenticator to communicate with a RADIUS server such as building RADIUS-REQUEST packets, encapsulating EAP messages and packet integrity checking etc. The RADIUS client module simply acquires a UDP socket for communicating to the RADIUS server.

The EAP state machine module is the core of this authentication simulator. It defines some global variables, timers and different state machines according to the state diagram defined in 802.1X, with minor changes. These state machines are used in two main programs, the authenticator and supplicant simulators.

- **Authenticator Simulator**

The authenticator simulator can be divided into five state machine components which are defined according to their definitions in the 802.1X protocol. They are Port Timers,

Reauthentication Timer, Authenticator Port Authentication Entity (PAE), Backend Authenticator and Key Transmit state machines.

The Port Timers state machine is responsible for decreasing timer variables every second. These timer variables are used for controlling the state operations of the authenticator and supplicant. The following figure (from IEEE 802.1X port-based network access control) shows the state diagram of the Port Timers state machine.

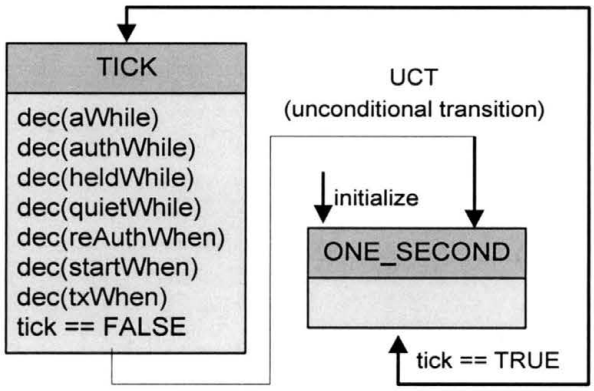

Figure 4-3 Port Tmers State machine

The Reauthentication Timer machine ensures that periodic reauthentication of the supplicant takes place. When the timer expires, it will signal the Authenticator PAE state machine to abort the supplicant's authentication status and reset the port status. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Reauthentication Timer state machine.

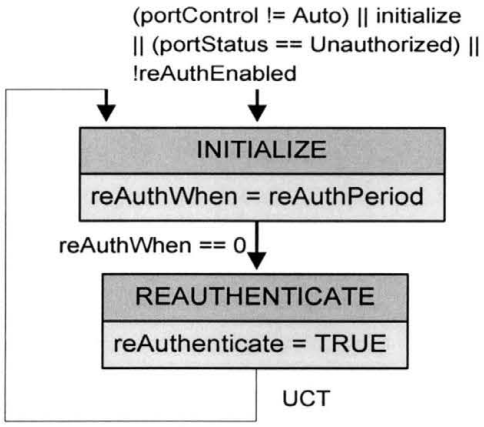

Figure 4-4 Reauthentication Timer State machine

The Authenticator PAE is in charge of controlling port status and requesting user identity when it senses a supplicant requests a port. It will signal the Backend Authenticator state machine to start a process, once it receives user identity, and it will be a conduit for the supplicant to Backend Authenticator conversation. According to the result from the Backend Authenticator, Authenticator PAE will decide the supplicant's port status and allow traffic to pass through. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Authenticator PAE state machine.

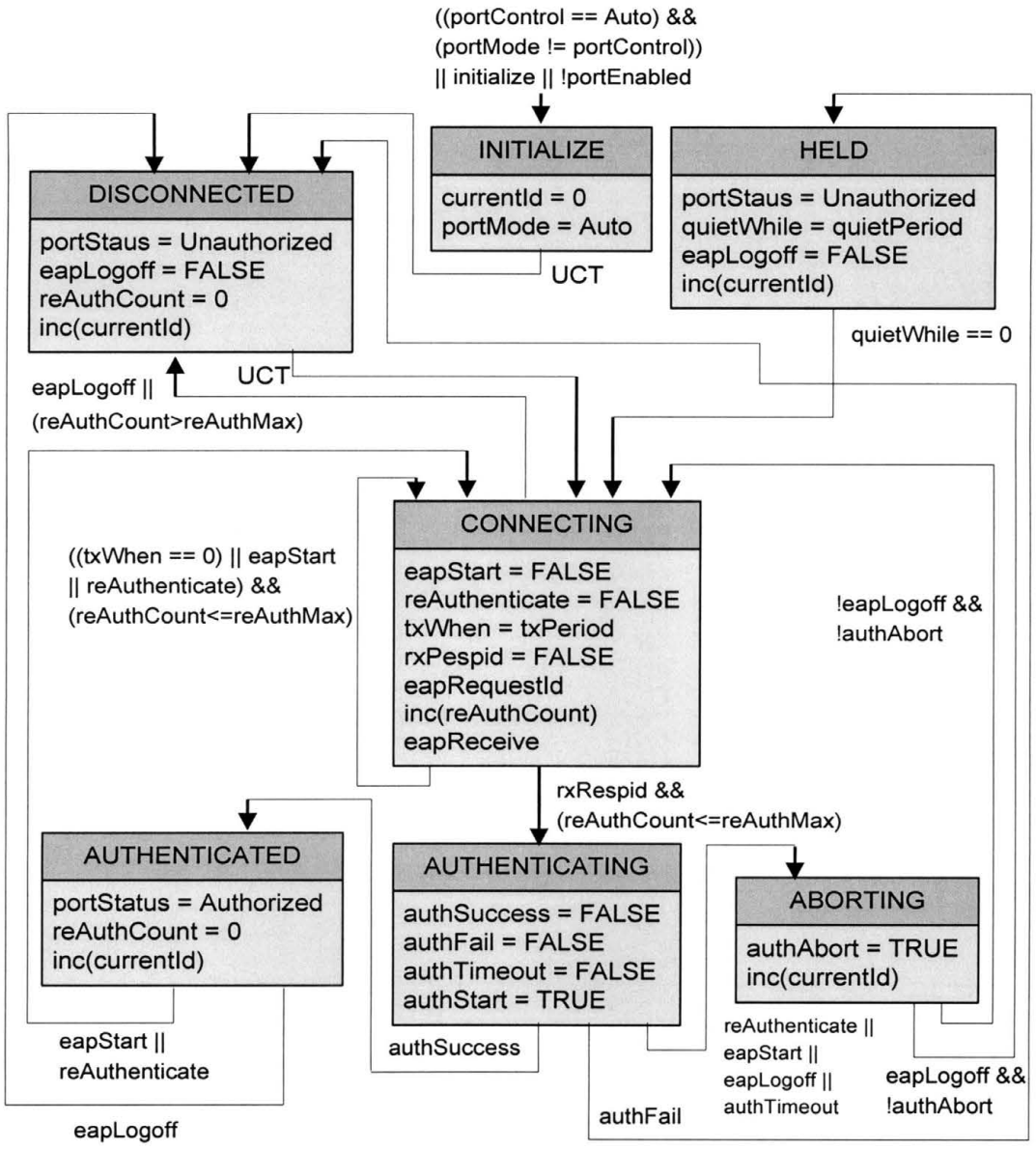

Figure 4-5 Authenticator PAE State machine

The Backend Authenticator state machine is responsible for communicating with the server and sending results back to supplicant once Authenticator PAE signals starting the authentication process. After the authentication process completes, it is also responsible for informing the Authenticator PAE, so that it can determine the supplicant's port status. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Backend Authenticator state machine.

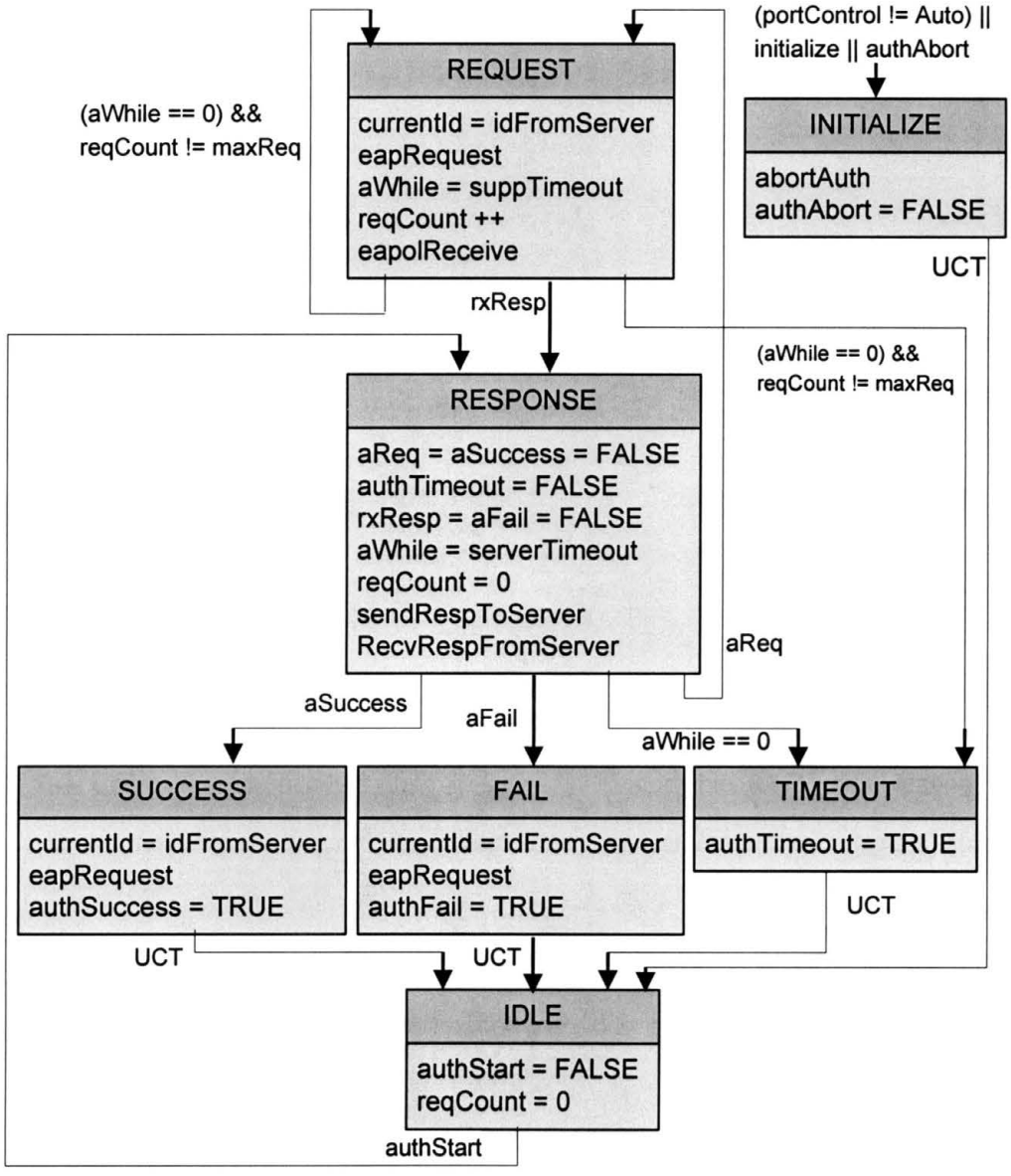

Figure 4-6 Backend Authenticator State machine

The Key Transmit state machine is activated when port status is authorized and key material is available. It will transmit the key information using EAPOL-KEY frame to supplicant. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Key Transmit state machine.

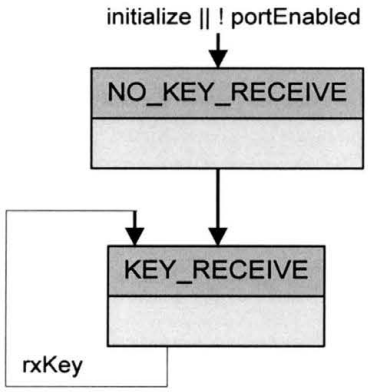

Figure 4-7 Key Transmit State machine

- **Supplicant Simulator**

The supplicant simulator can be configured to use different EAP authentication modules and is composed of three state machines: Port Timers, Key Receive and Supplicant PAE state machines.

The Key Receive state machine allows EAPOL-KEY frame to be received from the authenticator. After receiving keys, the supplicant can procede with any encryption mechanism negotiated between supplicant and authenticator. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Key Receive state machine.

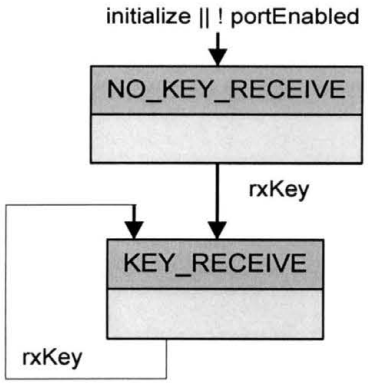

Figure 4-8 Key Receive State machine

The Supplicant PAE state machine is the main part of the supplicant simulator. It connects to the authenticator's port and triggers the authentication process by sending an EAPOL-START frame. It also provides any responses for EAP authentication requests coming from the server. The following figure (from IEEE 802.1X port-based network access control) indicates the state diagram of the Supplicant PAE state machine.

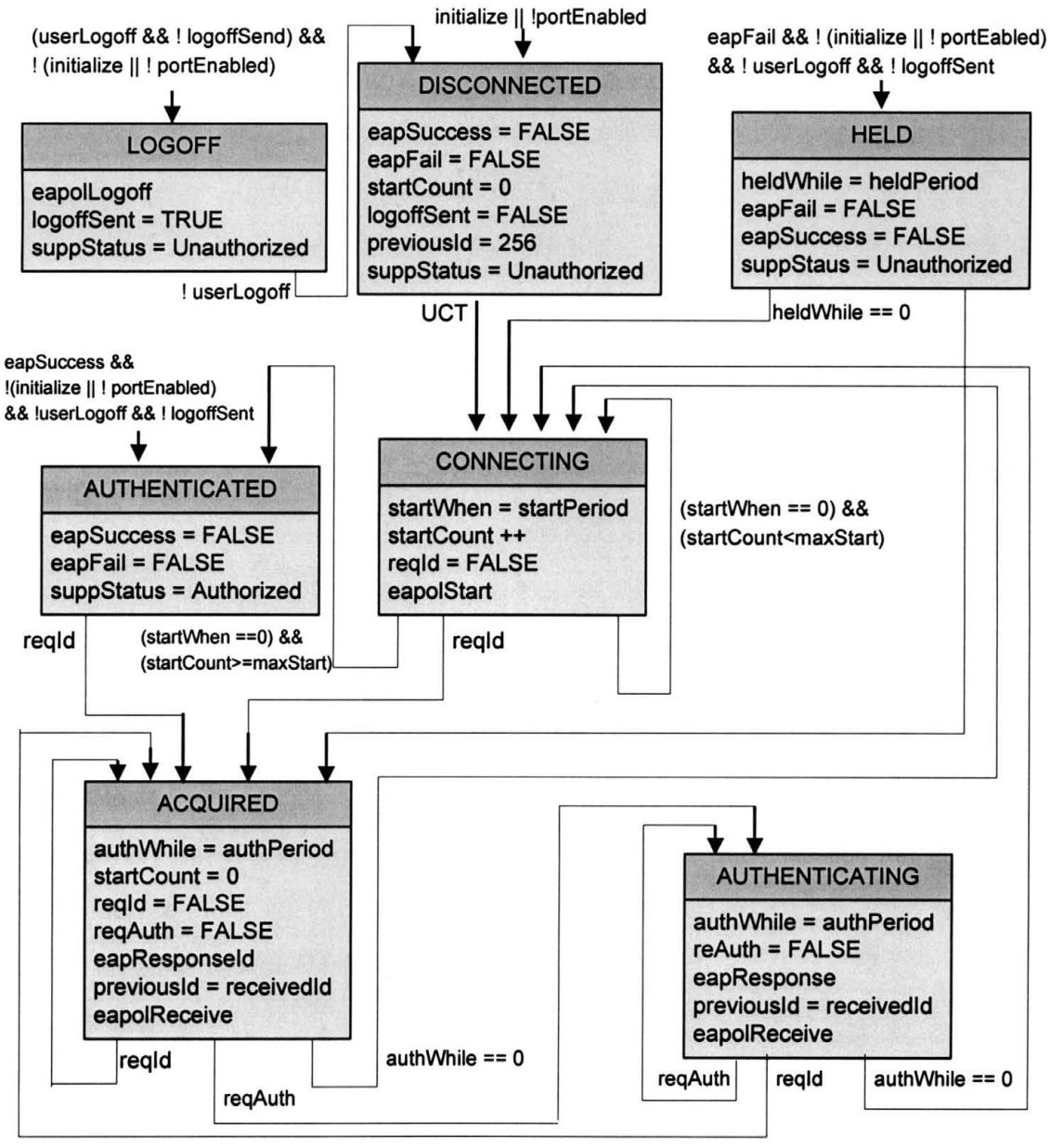

Figure 4-9 Supplicant PAE State machine

4.4. Supported Functionality

The functionality of the simulator can be described as follows:

- **802.1X capable without hardware support**

  The simulator implements the 802.1X protocol. It has EAP and RADIUS modules that are compatible with EAP and RADIUS protocols. Combined with a RADIUS server, it can simulate a 802.1X framework with different EAP authentication methods. Because it uses EAP over message queues, it does not need any wireless hardware devices. Since the simulator only provides authentication simulation, it only procedes until either authentication success or failure is attained, without any further data transmission.

- **Platform independent and work with real, external RADIUS server**

  Since this simulator is implemented using System V IPC message queues, any UNIX like system that support message queues can run this simulator. The simulator also has been tested with a real, external RADIUS server, FREERADIUS. Once both sides have been configured to use the same EAP authentication method, the simulator can perform authentications successfully using FREERADIUS.

- **Support EAP-MD5, EAP-TLS and EAP-TTLS**

  The simulator has implemented some common EAP authentication methods. It supports EAP-MD5 by requesting a user to enter a password from a command prompt. For certificate based authentication, the user needs to predefine certificates before the authentication process can start. The simulator does not support certificate management. For EAP-TTLS authentication, this simulator only supports inner method EAP-MD5, but any reasonable authentication method can be easily added as an inner method, once it is defined.

- **Easy implementation framework for new EAP authentication methods**

  One of the ideas of creating this simulator is to provide easy testing of new EAP authentication methods. As the design figure shows, all the EAP authentication modules rely on the same EAP software modules, which provide the basic function of EAP communication. One can implement a new EAP authentication method module simply by adding to the frame

format and functions provided in the EAP module without changing the software of the simulator.

4.5. Example of Simulation Output

The following example outputs demonstrate the simulation results for EAP-TLS authentication, using the authentication simulator with a real radius server, FREERADIUS.

```
  rlm_eap: Request found, released from the list
  rlm_eap: EAP/tls
  rlm_eap: processing type tls
  rlm_eap_tls: Authenticate
  rlm_eap_tls: processing TLS
rlm_eap_tls: Received EAP-TLS ACK message
  rlm_eap_tls: ack handshake is finished
  eaptls_verify returned 3
  eaptls_process returned 3
********** recv-key **********
7d fc 08 3b e8 11 09 2d 6f c4 6c ea 6f dd 6f a3 9f 7e 9b 1b 83 de 14 40 d4 fa 39
 21 94 e8 01 11
  rlm_eap: Freeing handler
  modcall[authenticate]: module "eap" returns ok for request 5
modcall: group authenticate returns ok for request 5
Sending Access-Accept of id 6 to 127.0.0.1:32769
        MS-MPPE-Recv-Key = 0x7dfc083be811092d6fc46cea6fdd6fa39f7e9b1b83de1440d4f
a392194e80111
        MS-MPPE-Send-Key = 0xe88853887ed085d3b22bb5179bf450b6847a1abe61402af6008
c8b83d125abaa
        EAP-Message = 0x03060004
        Message-Authenticator = 0x00000000000000000000000000000000
        User-Name = "yc1005"
Finished request 5
Going to the next request
Thread 1 waiting to be assigned a request
--- Walking the entire request list ---
Cleaning up request 5 ID 6 with timestamp 4180879b
Nothing to do.  Sleeping until we see a request.
```

Figure 4-10 FREERADIUS Server Running Result (Partial)

Once EAP authentication process succeeds, the authentication server will issue ACCESS-ACCEPT with AAA-Key (encrypt using MPPE and store in MS-MPPE-Send and Recv Key attributes) and EAP-SUCCESS (in EAP-Message attribute) to the authenticator (figure 4-10).

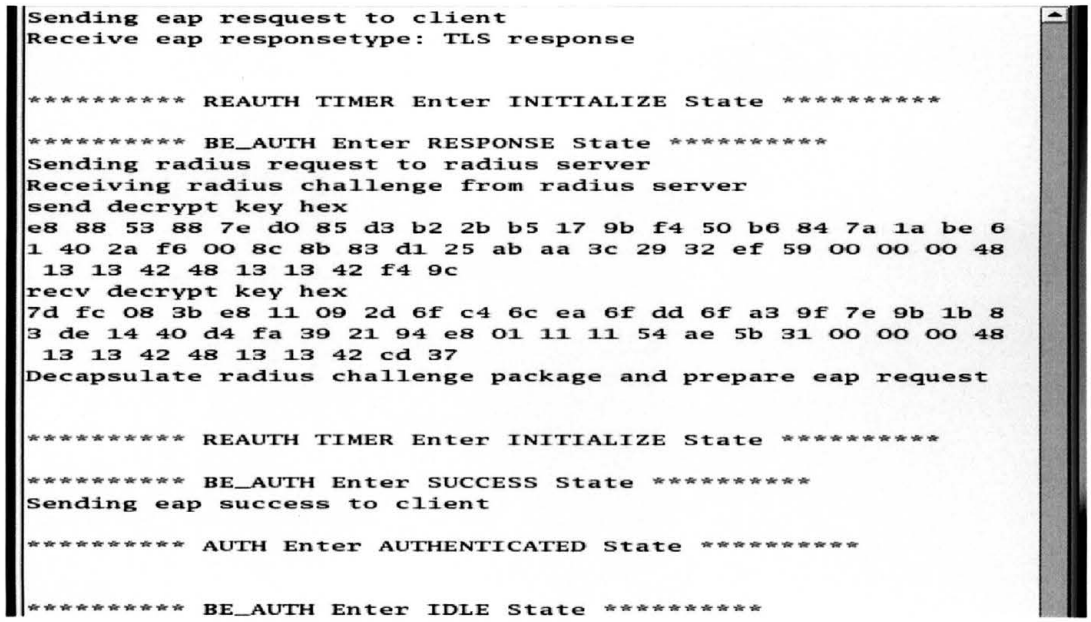

Figure 4-11 Authenticator Simulator Running Result (Partial)

The authenticator simulator will then decrypt MS-MPPE-Send and Recv Key attributes using its shared secret with the authentication server to get AAA-Key. Then it will forward EAP-SUCCESS to the supplicant simulator and enter authenticated state (figure 4-11).

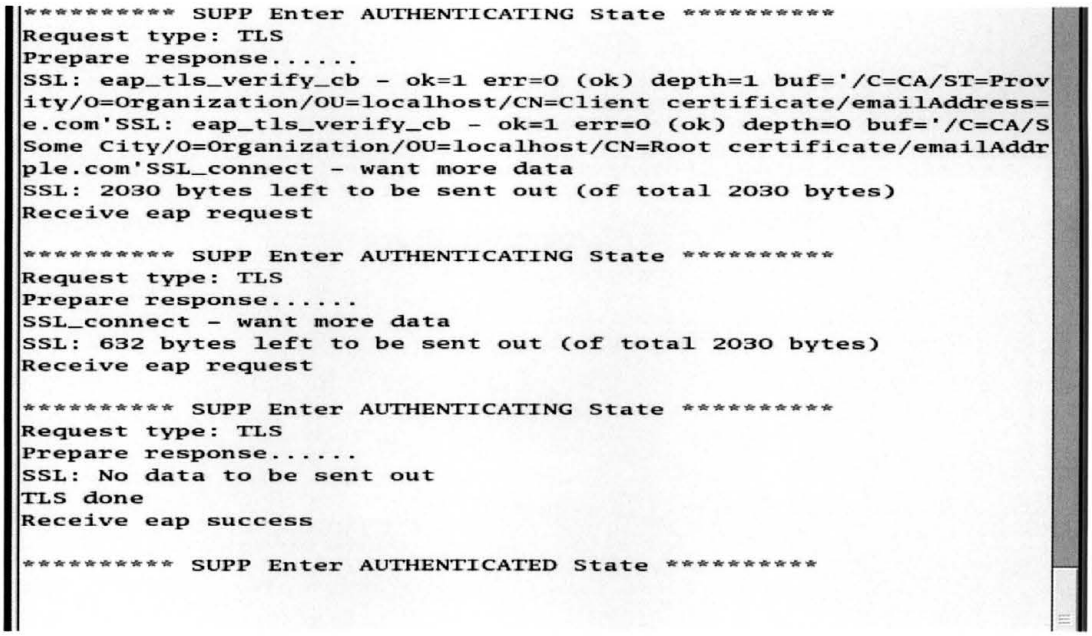

Figure 4-12 Supplicant Simulator Running Result (Partial)

Finally the supplicant simulator simply translates into authenticated state once it receives EAP-SUCCESS message (figure 4-12). The source code for the simulators is not included in the appendix because of the size – over eight thousand lines of code.

CHAPTER 5

A NEW 802.1X EAP AUTHENTICATION METHOD

5.1. Issues

EAP-TTLS and PEAP, mentioned in section 3.2.3, are both extensions of EAP-TLS. There are two reasons for using these two methods instead of EAP-TLS. First, by using EAP-TTLS or PEAP, it is only necessary to use digital certificates for the server to be authenticated to the client. Not requiring mutual certificate authentication simplifies certificate management, such as certificate distribution and revocation, etc. Also by setting up a secure channel first and using inner method for client authentication, identity protection is provided and eavesdropping is prevented, since the inner method packet is encrypted by TLS. Even if attackers can intercept EAP packets under wireless environments, they have no way of reading them. However, the draw back of using tunneled method is that it can allow man-in-the-middle attacks. PEAP, which uses EAP-MSCHAP-V2 as inner method, solves the man-in-the-middle attack by binding cryptographic keying material generated in EAP-MSCHAP-V2 with TLS keying material to produce a fresh AAA-key. On the other hand, EAP-TTLS may be able to prevent man-in-the-middle attacks depending on the inner method chosen. In this chapter, we focus on the EAP-TTLS by choosing EAP-MD5 as its inner method and propose an extension of it, EAP-TTLS-CHENG to solve man-in-the-middle attacks. Then we test the extension by adding its features to the simulator and by modifying the publicly available FREERADIUS server.

5.2. EAP-TTLS with EAP-MD5 Authentication Procedure

As the Chapter 3 EAP-TTLS section mentioned, EAP-TTLS has a two phase authentication. The following figure gives a detailed message exchange of EAP-TTLS:

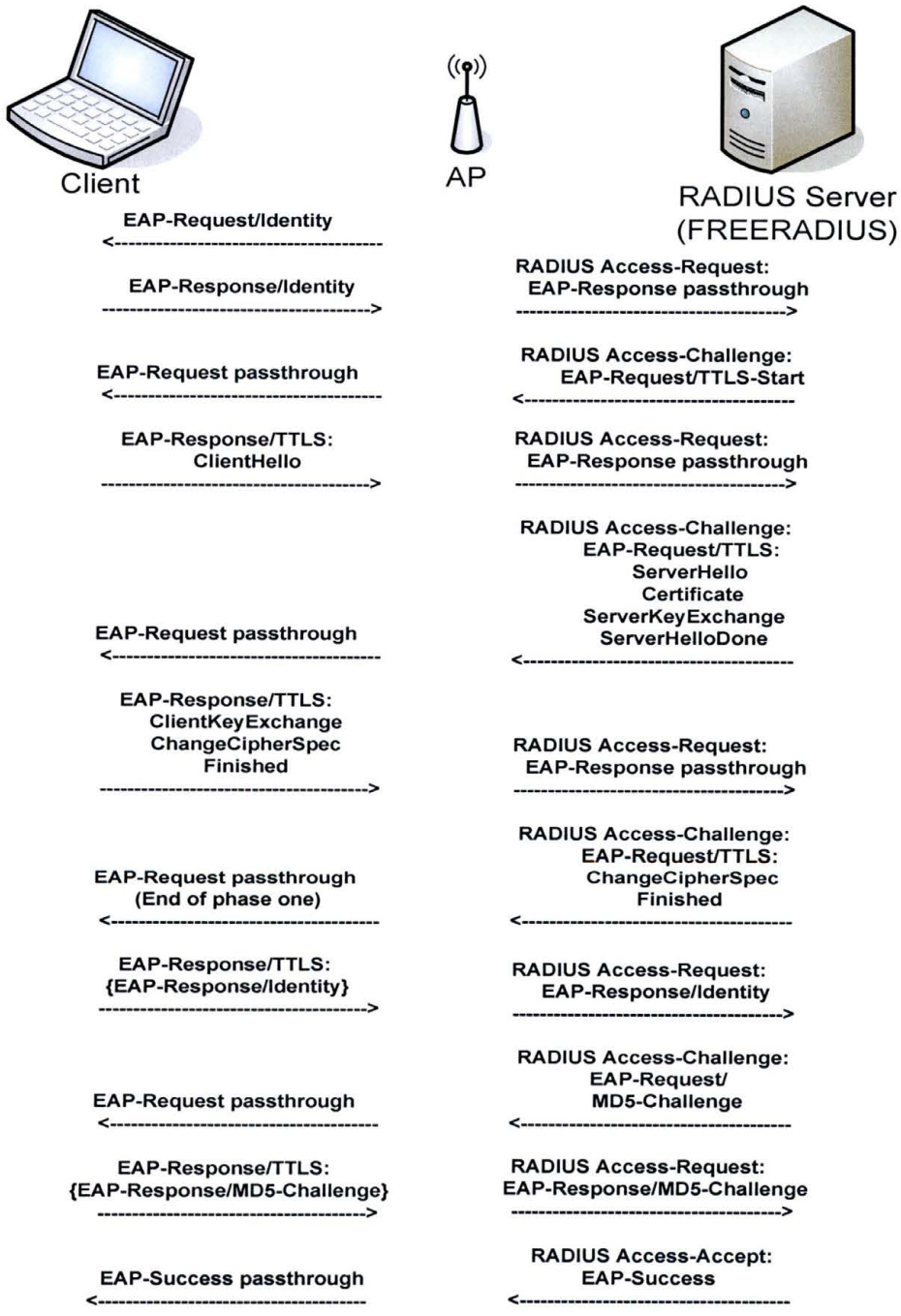

Figure 5-1 EAP-TTLS Message Exchange

Phase one establishes a secure channel by encapsulating the TLS handshake inside EAP-TTLS packets and then forwarding it to the RADIUS server by encapsulating EAP inside RADIUS. After both parties perform the TLS KeyExchange handshake, the TLS negotiated ciphersuite is used to set up a protected channel for phase two EAP conversations. Starting from phase two, EAP-TTLS uses Attribute-Value-Pairs (AVPs), which are compatible with RADIUS attributes format, to carry packets, and encapsulate AVPs inside TLS. TLS is then again encapsulated inside EAP-TTL and forwarded to the server inside RADIUS packets. The following figure indicates the packet relationship in the EAP-TTLS phase one and phase two:

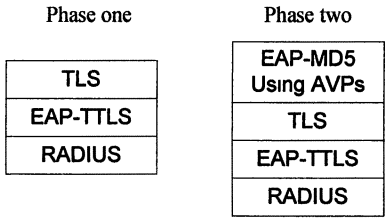

Figure 5-2 Packet Relationship in EAP-TTLS

If a client is configured to use anonymous identity in the phase one, the client will send its real identity to AP in the beginning of the phase two. After the identity is received by the server, the server will issue an EAP-MD5 challenge to the client. By sending the correctly computed MD5 hash value of the challenge back to server, the authentication process is completed. As was previously described in Chapter 3, at this point, both parties will take the TLS Master-Secret derived from TLS handshake exchange together with TLS Server-Random, TLS Client-Random and label - "ttls keying material", as inputs and produce a 64 octet Master-Session-Key. This 64 octet Master-Session-Key is then used by the server as AAA-key to pass to authenticator using MPPE. The AAA-Key that is passed to the authenticator can be divided into two halves. They are 32 octets “Peer to Authenticator Encryption Key” (Enc-RECV-Key) and 32 octets “Authenticator to Peer Encryption Key” (Enc-SEND-Key), which are transported separately in MS-MPPE-Recv-Key and MS-MPPE-Send-Key attributes in ACCESS-ACCEPT packet; Once client and authenticator both possess AAA-Key, they can verify each party's identity and compute fresh Transient-Session-Key for data encryption.

5.3. Man-in-the-Middle Attack Scheme

Surprisingly, although EAP-TTLS seemed to provide better security protection by running client authentication process inside a secure tunnel, it is still open to man-in-the-middle-attacks. The reason is the inner authentication method is not aware of the existence of the secure channel. The man-in-the-middle attack scheme can be described as the following figure indicates:

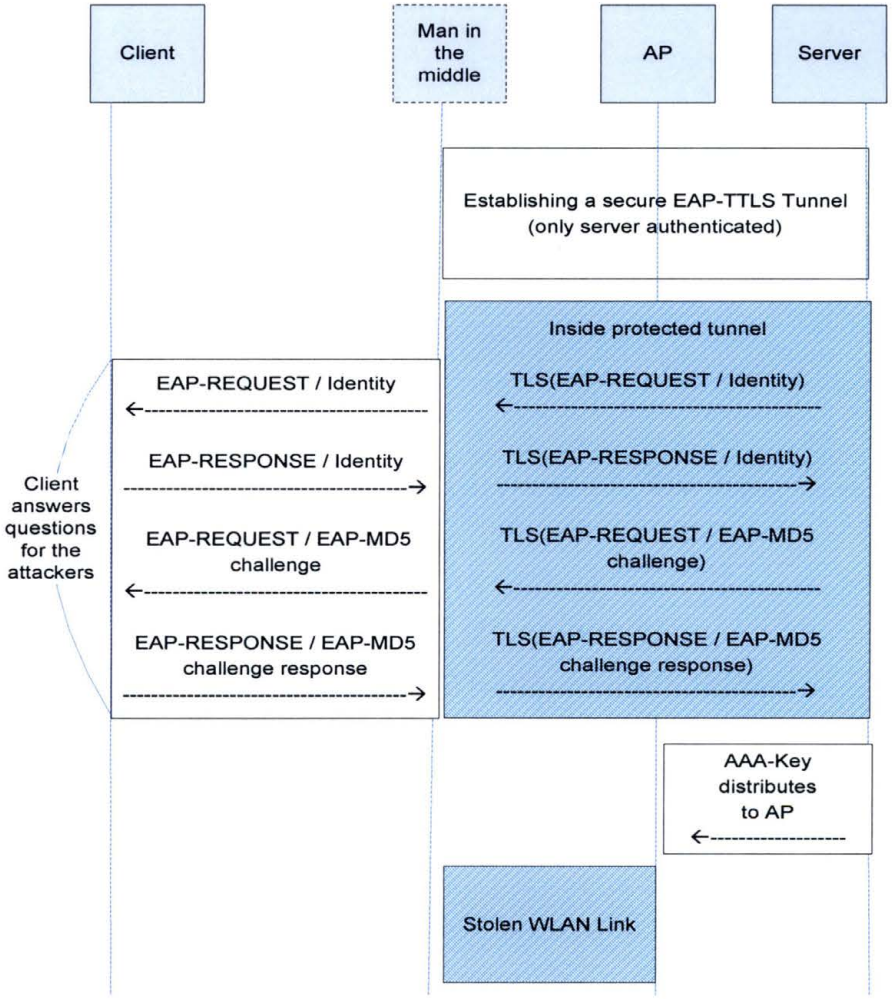

Figure 5-3 Man-in-the-Middle for EAP-TTLS/EAP-MD5

Since, in phase one, only the server is required to authenticate to client without the user providing its real identity, anonymous attackers can perform a TSL handshake and give an anonymous identity, establishing a secure tunnel with the server before the real client contacts the server. Once an attacker succeeds in establishing a tunnel with the server, it can pretend to be an AP and request the real client to perform client authentication by first asking client's identity. After

receiving the identity from the real client, attackers can encapsulate it and send it back to the server via the secure tunnel. With EAP-MD5 configured as inner method, the server will then issue an EAP-MD5 challenge to the attacker. The attacker can simply forward the MD5 challenge as EAP-REQUEST to the real client. Since the client has no knowledge of the tunnel, it will think the server is requesting EAP-MD5 authentication. It then will compute the MD5 hash value of the challenge with its shared secret and send it back to the attacker. By forwarding the response to the server, the authentication process is completed. In addition, EAP-TTLS only uses keying material generated from the TLS handshake to compute a Master-Session-Key using TLS pseudo-random function (PRF). Therefore once the server distributes AAA-Key to AP, both attacker and AP are able to prove possession of the AAA-Key. At this point, the attacker can steal the port by issuing EAP-FAIL to real client, denying its request and then connect to the port with the stolen identity.

5.4. EAP-TTLS-CHENG

As the attack scenario mentioned above demonstrates, we can conclude there are two reasons why man-in-the-middle attacks can succeed. First, a client will compute the hash value for attackers without knowing the existence of tunnel. Second, EAP-TTLS only uses TLS keying material to compute the key. We can not solve the first problem without changing the standard definition of EAP-MD5. Therefore, we counter the second problem by combing the EAP-MD5 shared secret with TLS keying material to produce a new Master-Session-Key. The idea is, instead of using the original TLS PRF which takes TLS Master-Secret, TLS Server-Random, TLS Client-Random and label - "ttls keying material" as inputs to generate Master-Session-Key, we use the following modified PRF function to compute 64 octets Master-Session-Key.

Master-session-key = PRF-CHENG (TLS Master-Secret, label - “ttls keying material with inner md5”, TLS Server-Random, TLS Client-random, shared secret (userpassword)).

With this scheme, even though an attacker can be authenticated successfully to the server, it does not have a user password to compute the Master-Session-Key. Once the server distributes

AAA-Key to the AP, the attacker can no longer prove its possession of the AAA-Key to the AP. Only the real client can make the calculation, since only the real client has the shared secret. The following example output demonstrates the simulation result of two clients: one (man-in-the-middle) use regular EAP-TTLS and the other one (true supplicant) use EAP-TTLS-CEHNG with radius server using EAP-TTLS-CHENG. (Highlight indicates the key generation results.)

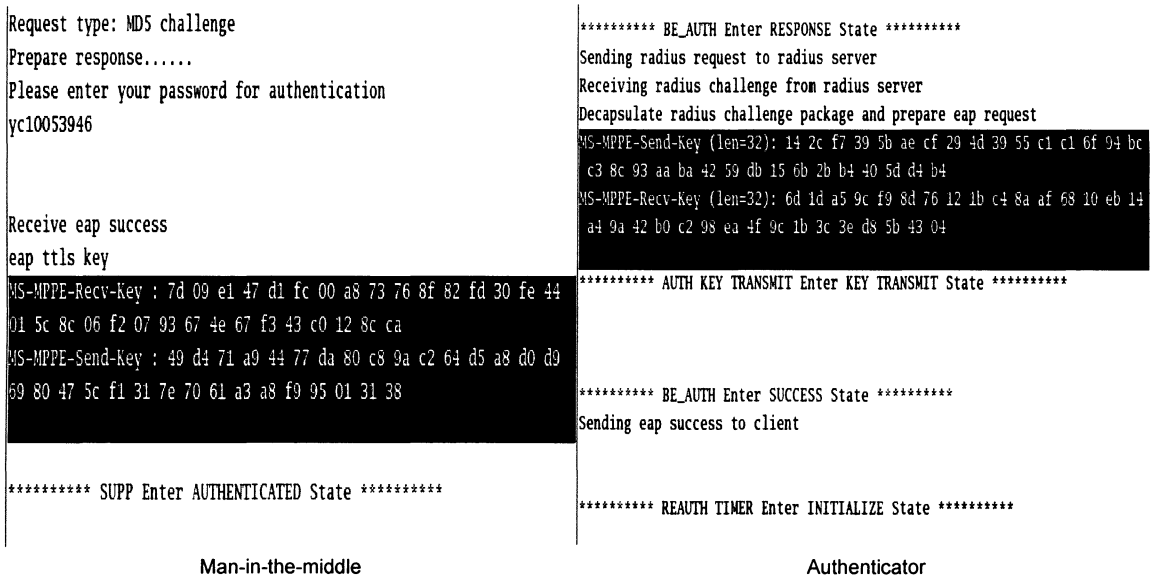

Figure 5-4 Man-in-the-Middle vs. Authenticator simulation result

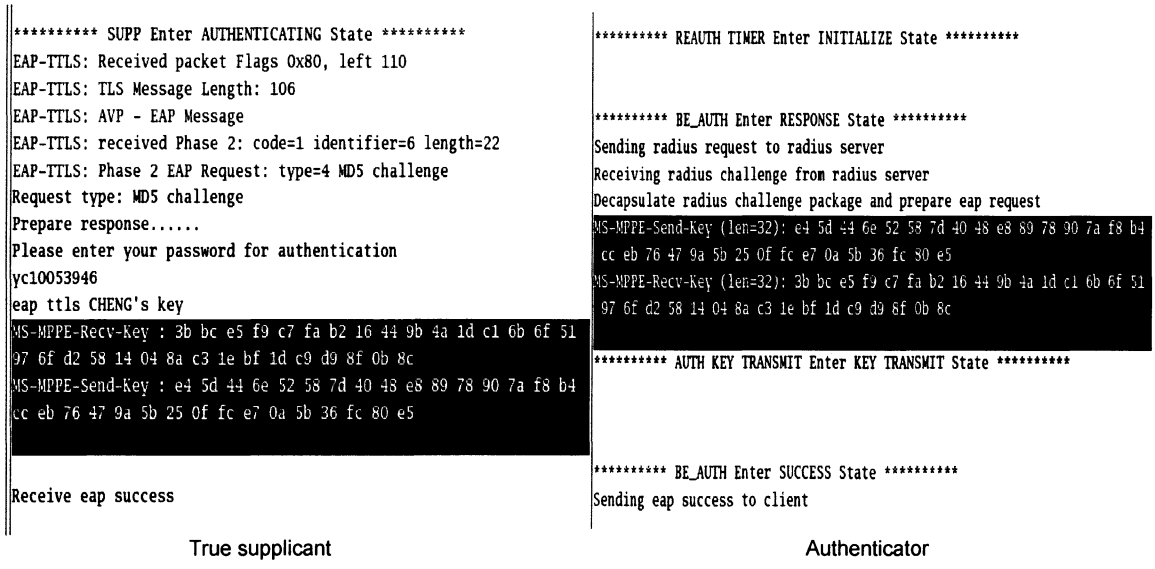

Figure 5-5 True supplicant vs. Authenticator simulation result

As the above result of the simulation process indicates, we can see even though the man-in-the-middle can authenticate with server successfully; it generates a different AAA-Key compared with the one the authenticator received (figure 5-4). On the other hand, the true supplicant, which uses EAP-TTLS-CHENG, generates the same AAA-Key as the authenticator (figure 5-5). Therefore, we can conclude that combining the user password with the key generation function, the man-in-the-middle attack can be solved, since the attacker can not prove possession of the same AAA-Key to the authenticator. By using the password as part of the keying material, it is possible for an attacker to mount a dictionary attack against the password so that he can get enough information to compute the same key. But this possibility is really low if the user uses a strong password. Also by emulating CHENG's method, legacy authentication methods such as CHAP or MSCHAP, used as an inner method of EAP-TTLS, can be modified by using a new key generation function to prevent man-in-the-middle attacks.

CHAPTER 6

LATEST DEVELOPMENT IN STANDARDS

6.1. Wi-Fi Protected Access (WPA)

The original 802.11 standard was created by the IEEE in 1997. The IEEE makes an effort to update and correct the standard, but there are still some areas that are ambiguous or not fully defined, creating difficulties in terms of design issues. The Wi-Fi Alliance [1] was formed by a group of major manufactures. They created a Wi-Fi certification program for testing manufacturer's products. In order to quickly provide a security solution for 802.11's WEP key weakness without waiting for new 802.11i standard, a new security specification was adopted by Wi-Fi Alliance in 2002. Wi-FI Protected Access [35] is a subset of the incoming 802.11i standard and is compatible with all 802.11 standard devices. WPA can be used in two environments: WPA-Enterprise which combines 802.1X EAP authentication framework with the Temporary Key Integrity Protocol [28] [29](TKIP) for strong encryption; WPA-Personal/SOHO which uses Pre-Shared Key (PSK) that allows users to manually enter passwords for authentication was combined with TKIP for data encryption.

TKIP was created to solve the weakness of WEP. It adds several features: TKIP uses 128-bit keys and employs a key hierarchy and key management to mathematically derive and change encryption keys and IV values automatically (creating per-packet keys). It also uses 64-bit

[1] The Wi-Fi Alliance is a nonprofit international association formed in 1999 to certify interoperability of wireless local area network products based on IEEE 802 11 specifications

Message Integrity Check (MIC or called "Michael [27]") to prevent packet forgeries. The Initialization Vector (IV), one of the big weaknesses of WEP, is solved by increasing the size of IV (using 48-bit IV). The following figure (source [34]) demonstrates the WPA encryption process using TKIP and Michael.

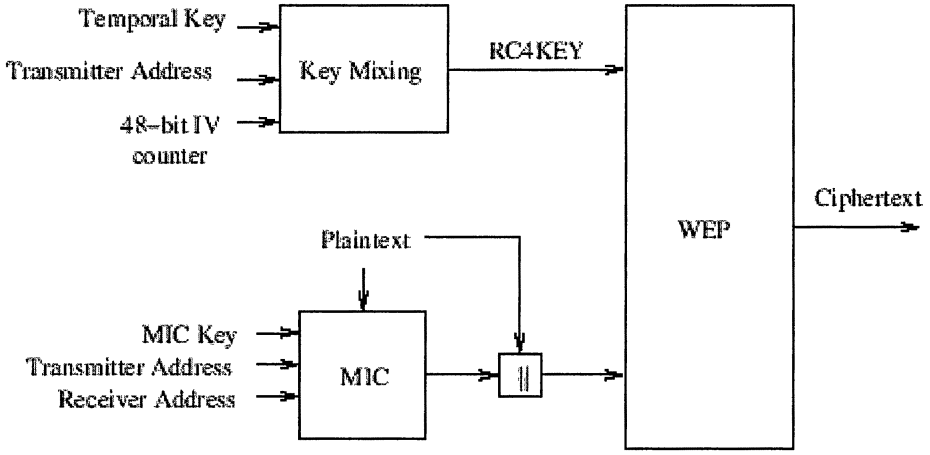

Figure 6-1 WPA Encapsulation Process

As figure 6-1 indicates, TKIP defined in [29] uses 16-byte Temporal Key (TK) together with 6-byte Transmitter Address and a 48-bit IV as inputs to Key Mixing function and outputs a 16-byte RC4 key. The Key mixing function is a two-phase process. There is phase one, P1K = Phase1 (TK, TA, IV32) where IV32 is the 32-most significant bit of 48-bit IV; and phase two, RC4KEY = Phase2 (P1K, TK, IV16) where IV16 is the least significant 16 bits of the 48-bit IV. A recent paper [34] has pointed out the weakness in the TKIP. Since TKIP security depends on RC4-keys, if the attacker is able to get a few RC4-keys computed under the same IV32, it is possible for an attacker to compute backward through Phase2 to recover the TK and thus, decrypt any packet the same way the receiver does.

6.2. WPA2 and 802.11i Standard

In the 802.11i [3] standard, an important component was defined: Robust Secure Network (RSN), which uses an Advance Encryption Standard [31] (AES) cipher system and Counter Mode CBC MAC protocol [12] (CCMP) along with 802.1X and EAP. RSN is a protocol for

establishing secure communication over 802.11 wireless networks. By default, it requires use of AES but currently, in order to provide backward compatible to TKIP and WEP, 802.11i also defines a Transitional Security Network (TSN) so that both RSN and WEP systems can operate in parallel. WPA2 is the second generation of WPA and is based on the final 802.11i amendment to the 802.11 standard. The main differences between WPA and WPA2 are WPA2s use RSN and AES/CCMP instead of TKIP, which is still a RC4 based encryption algorithm in WPA. WPA2 can also be used in Enterprise and Personal environments and is backward compatible to WPA. AES is a complex and strong encryption mechanism and it uses a block cipher which operates with blocks of 128-bit data. CCMP is a security protocol based on AES. It uses Counter Mode with CBC-MAC (CCM) mode of operation and computes a Message Integrity Check based on CBC-MAC method. CCMP use a 48-bit IV same as TKIP to ensure the lifetime of AES key. Unlike TKIP, CCMP doesn't need per-packet keys. It just uses the same AES key to provide confidentiality and integrity protection. The following figure summaries features of WEP vs. TKIP&CCMP (source: WPA-strong standards-based, interoperable security for today's Wi-Fi networks).

	WEP	TKIP	CCMP
Cipher Key Size(s)	RC4 40- or 104-bit encryption	RC4 128-bit encryption, 64-bit authentication	AES 128-bit
Key Lifetime Per-packet key	24-bit wrapping IV Concatenate IV to base key	48-bit IV TKIP mixing function	48-bit IV Not needed
Integrity Packet Header	None	Source and destination addresses protected by Michael	CCM
Packet Data Replay detection	CRC-32 None	Michael Enforce IV sequencing	CCM Enforce IV sequencing
Key Management	None	IEEE 802.1X	IEEE 802.1X

Figure 6-2 WEP vs. TKIP & CCMP

Whether using TKIP or AES/CCMP, 802.1X is the core requirement for WPA and 802.11i since it provides centralized control and dynamically generates and distributes the keying materials. TKIP and AES/CCMP are used for providing a secure data encryption between supplicant and AP after the authentication process is complete. Both TKIP's Temporal Key and AES/CCMP rely

on the keying material generated from the 802.1 X EAP authentication methods such as EAP-TLS or EAP-TTLS. By countering the man-in-the-middle attacks in EAP-TTLS using CHENG's variation, it directly increases the safety of using TKIP and AES/CCMP since their keying material is derived from the EAP authentication method. Also testing different EAP authentication methods using the simulator we developed helps us analyze the security problems they encounter and solve them in advance to ensure the security of the TKIP and AES/CCMP encryption.

TOOLS AND SOFTWARE USED

- Red Hat Linux – http://www.redhat.com
- Windows XP Professional- http://www.microsoft.com
- Free RADIUS – http://www.freeradius.org/
- HostAP – http://hostap.epitest.fi/
- WPA_Supplicant – http://hostap.epitest.fi/
- Openssl - http://www.openssl.org/

APPENDICES

A.1. FREERADIUS Set Up

In order to use FREERADIUS for EAP-TLS authentication, Openssl has to be installed.

Install Openssl:

1. Download Openssl snapshot from ftp://ftp.openssl.org/snapshot/
2. Once you untar it and change directory to the openssl directory.
3. In the command prompt, type:
   #./config shared --prefix=/usr/local/openssl
   # make
   # make install
4. If no error messages show up, installation process is successful.

Install FREERADIUS:

1. Download FREERADIUS from www.freeradius.org
2. Untar and change to freeradius directory.
3. In the command prompt, type:
   #./configure --with-openssl-includes=/usr/local/openssl/include \
   --with-openssl-libraries=/usr/local/openssl/lib \
   --prefix=/usr/local/radius
   # make
   # make install
4. Once no error showup, installation succeeds.

Configure FREERADIUS:

After successful installation, there are three file has to be modified in order to run radius for different authentication. They are eap.conf, clients.conf and users which are located at /usr/local/radius/etc/raddb.

1. clients.conf:
   This file contains information about the authenticator (AP). The example configuration is listed as follows:

   ```
   client 127.0.0.1 {
                secret = aaaaabbbbb
                shortname = localhost
   }
   ```

   where "127.0.0.1" is the IP address where your authenticator simulator is. In above example, authenticator simulator and FREERADIUS server is located in the same machine.
   "secret = aaaaabbbbb" is the share secret used between authenticator simulator and server.
   "shortname = localhost" is the name for your authenticator simulator. It can be any name you want to call.

2. users
   This file contains information about the supplicant who wants to authenticate with RADIUS server. Example is listed as following:

   ```
   # EAP-MD5
   yc1005 Auth-Type :=EAP, User-Password == "yc10053946"
   ```

```
# EAP-TLS
yc1005
```

where "yc1005" is the user name wish to be authenticated by server.
"yc10053946" is the user password for user.

As above example, for EAP-MD5 authentication username, auth-type and user-password have to be included in the users file. For EAP-TLS, only user name has to be included. Once you have EAP-MD5 and TLS configuration, you don't have to add any user configuration for EAP-TTLS if inner method uses EAP-MD5.

3. eap.conf
This file contains eap authentication type information. It defines different eap authentication type that can be used for RADIUS server. The configuration example is listed as below:

```
eap {
    default_eap_type = md5

    md5 {
    }
    #tls {
            #private_key_password = whatever
            #private_key_file = ${raddbdir}/certs/cert-srv.pem

            #certificate_file = ${raddbdir}/certs/cert-srv.pem
            #CA_file = ${raddbdir}/certs/demoCA/root.pem

            #dh_file = ${raddbdir}/certs/dh
            #random_file = ${raddbdir}/certs/random

            #fragment_size = 1024
    #}
    #ttls {

            #default_eap_type = md5
    #}
```

As above example, "default_eap_type = md5" decides which authentication method is used by radius server. The default set up in eap.conf is md5. If you want to use TLS or TTLS you have to uncomment the TLS section and TTLS section first as above example shows. In the TLS section, you have to give the path to different certificates where "private_key_password" is the password you use to decode "private_key_file", "private_key_file" stores your private key, "certificate_file" stores your public key certificate, "CA_file" stores your CA's certificate, "dh_file" and "random_file" both store random information. In the TTLS section, it uses the same certificate information in TLS section. Therefore, once the configuration of the certificate is done in the TLS section, you don't need to configure any certificate path. The only thing you need to do is to give a default inner method type as above example.

Run FREERADIUS:

You can run radius server by typing
#/usr/local/radius/sbin/radiusd
Or run it with debug mode by typing
#/usr/local/radius/sbin/radiusd -xxx

A.2. Generate Certificate for EAP-TLS/TTLS

To use EAP-TLS/TTLS, server and client certificates are needed. Since we didn't go out to the certificate authority to get a public certificate, we can use openssl to generate a self-signed up CA certificate and sign our own certificate. A script file, CA.all, provided by freeradius can be used to easily generate server and client certificate. This is file is located in "script" directory under freeradius. However, before running CA.all script file, openssl has to be configured by updating the openssl configuration file, openssl.conf, which is located at /usr/local/openssl/ssl. The following example demonstrates the partial of configuration that has to be changed for openssl.

Openssl.conf:

```
countryName = Country Name (2 letter code)
countryName_default = US
countryName_min = 2
countryName_max = 2

stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default = Texas

localityName = Locality Name (eg, city)
localityName_default = San Marcos

0.organizationName = Organization Name (eg, company)
0.organizationName_default = TXSTATE

organizationalUnitName = Organizational Unit Name
organizationalUnitName_default = CS

commonName = Common Name (eg, YOUR name)
commonName_max = 64
commonName_default = CA

emailAddress = Email Address
emailAddress_max = 40
emailAddress_default = yc1005@txstate.edu

# SET-ex3 = SET extension number 3

[ req_attributes ]

challengePassword = A challenge password
challengePassword_min = 4
challengePassword_max = 20
challengePassword_default = whatever

unstructuredName = An optional company name
```

Once you update the openssl.conf file, you can run CA.all file. However, you might need to change the path of the openssl in CA.all script depending on your openssl installation. When you run CA.all script, the above openssl configuration information will appear for three times. The first pass of information will produce CA's root certificate. If you set up your openssl configuration file same as above, you only need to accept all the settings for the first pass. The second pass will generate client's certificate signed up by CA's certificate. You only need to change "commonName" option which is CA as above example to the client name you want to use. The last pass of information will result server certificate. Same as client pass, you only need to

change a “commonName” to a server name without changing anything else. The following gives an example of the CA.all file that I used to generate certificates.

CA.all

```
SSL=/usr/local/openssl/

export PATH=${SSL}/bin/:${SSL}/ssl/misc:${PATH}

export LD_LIBRARY_PATH=${SSL}/lib

rm -rf demoCA roo* cert* *.pem *.der

echo -e ""
echo -e "\t\t####################"
echo -e "\t\tcreate private key"
echo -e "\t\tname : name-root"
echo -e "\t\tCA.pl -newcert"
echo -e "\t\t####################\n"

openssl req -new -x509 -keyout newreq.pem -out newreq.pem -days 730 -passin pass:whatever -
passout pass:whatever

echo -e ""
echo -e "\t\t####################"
echo -e "\t\tcreate CA"
echo -e "\t\tuse just created 'newreq.pem' private key as filename"
echo -e "\t\tCA.pl -newca"
echo -e "\t\t####################\n"

echo "newreq.pem" | /usr/local/openssl/ssl/misc/CA.pl -newca

#ls -lg demoCA/private/cakey.pem

echo -e ""
echo -e "\t\t####################"
echo -e "\t\texporting ROOT CA"
echo -e "\t\tCA.pl -newreq"
echo -e "\t\tCA.pl -signreq"
echo -e "\t\topenssl pkcs12 -export -in demoCA/cacert.pem -inkey newreq.pem -out root.pem"
echo -e "\t\topenssl pkcs12 -in root.cer -out root.pem"
echo -e "\t\t####################\n"

openssl pkcs12 -export -in demoCA/cacert.pem -inkey newreq.pem -out root.p12 -cacerts -passin
pass:whatever -passout pass:whatever
openssl pkcs12 -in root.p12 -out root.pem -passin pass:whatever -passout pass:whatever
openssl x509 -inform PEM -outform DER -in root.pem -out root.der

echo -e ""
echo -e "\t\t####################"
echo -e "\t\tcreating client certificate"
echo -e "\t\tname : name-clt"
echo -e "\t\tclient certificate stored as cert-clt.pem"
echo -e "\t\tCA.pl -newreq"
echo -e "\t\tCA.pl -signreq"
echo -e "\t\t####################\n"
```

```
openssl req -new -keyout newreq.pem -out newreq.pem -days 730 -passin pass:whatever -passout pass:whatever
openssl ca  -policy policy_anything -out newcert.pem -passin pass:whatever -key whatever -extensions xpclient_ext -extfile xpextensions -infiles newreq.pem

openssl pkcs12 -export -in newcert.pem -inkey newreq.pem -out cert-clt.p12 -clcerts -passin pass:whatever -passout pass:whatever
openssl pkcs12 -in cert-clt.p12 -out cert-clt.pem -passin pass:whatever -passout pass:whatever
openssl x509 -inform PEM -outform DER -in cert-clt.pem -out cert-clt.der

echo -e ""
echo -e "\t\t####################"
echo -e "\t\tcreating server certificate"
echo -e "\t\tname : name-srv"
echo -e "\t\tserver certificate stored as cert-srv.pem"
echo -e "\t\tCA.pl -newreq"
echo -e "\t\tCA.pl -signreq"
echo -e "\t\t####################\n"

openssl req -new  -keyout newreq.pem -out newreq.pem -days 730 -passin pass:whatever -passout pass:whatever
openssl ca  -policy policy_anything  -out newcert.pem -passin pass:whatever -key whatever -extensions xpserver_ext -extfile xpextensions -infiles newreq.pem

 openssl pkcs12 -export -in newcert.pem -inkey newreq.pem -out cert-srv.p12 -clcerts -passin pass:whatever -passout pass:whatever
openssl pkcs12 -in cert-srv.p12 -out cert-srv.pem -passin pass:whatever -passout pass:whatever
openssl x509 -inform PEM -outform DER -in cert-srv.pem -out cert-srv.der

echo -e "\n\t\t####################\n"
```

After successful running CA.all script, it will produce nine certificates. They are root.pem, root.p12, root.der, cert-clt.pem, cert-clt.p12, cert-clt.der, cert-srv.pem, cert-srv.p12, cert-srv.der. For freeradius server, it has to use root.pem and cert-srv.pem. By putting root.pem and cert-srv.pem to the directory path where tls section of eap.conf file indicates, FREERADIUS server will load certificates when you run it. For supplicant simulator, it has to use cert-clt.der, cert-clt.pem and root.pem. The configuration of loading certificates in the supplicant simulator is described in next section.

A.3. Authentication Simulator Set Up

The authentication simulator doesn't need to configure which authentication method it uses. It will depend on RADIUS server which means if you configure RADIUS to use EAP-MD5 authentication then authentication simulator will also run EAP-MD5. However, you do need to give certificate information if you use EAP-TLS or TTLS. To configure certificates for supplicant simulator, you have to look at the source code file, tls.c. In tls.c file, there are some #define statements where you can define your certificates. Once you are done with defining certificates, you can compile the source code and supplicant simulator will load certificates when it runs.

Compile Authentication simulator:

Type following command on the command prompt
make ap
make supplicant
Once you are done, you will get two executable file, ap and supplicant.

Run Authenticator simulator:

#./ap tsunami 127.0.0.1 aaaaabbbbb

where "tsunami" is the ssid of authenticator simulator.
"127.0.0.1" is the RADIUS server IP address
"aaaaabbbbb" is the shared secret between authenticator simulator and RADIUS server. This shared secret must be the same as the one defined in RADIUS clients.conf file.

Run Supplicant simulator:

#./supplicant tsunami yc1005

where "tsunami" is the ssid id of authenticator simulator
"yc1005" is the user name that used for authentication. This user name information must be the same as the one defined in RADIUS users file.

Use EAP-TTLS-CHENG:

In order to use EAP-TTLS-CHENG, there are two source code files in authentication simulator have to be modified. In eap.c and eap_ttls.c, you have to uncomment the "#define CHENG" statement and then recompile the source code. Also you have to do the same thing in FREERADIUS source file listed as next section and recompile the source code.

A.4. Modification of FREERADIUS Source Code

```
/*
 * rlm_eap_ttls.c  contains the interfaces that are called from eap
 *
 * Version:     $Id: rlm_eap_ttls.c,v 1.5 2004/03/05 17:51:17 aland 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.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Copyright 2003 Alan DeKok <aland@freeradius.org>
 */

#include "autoconf.h"
#include "eap_ttls.h"

//#define CHENG

typedef struct rlm_eap_ttls_t {
        /*
         *      Default tunneled EAP type
         */
        char    *default_eap_type_name;
        int     default_eap_type;

        /*
         *      Use the reply attributes from the tunneled session in
         *      the non-tunneled reply to the client.
         */
        int     use_tunneled_reply;

        /*
         *      Use SOME of the request attributes from outside of the
         *      tunneled session in the tunneled request
         */
        int     copy_request_to_tunnel;
} rlm_eap_ttls_t;


static CONF_PARSER module_config[] = {
        { "default_eap_type", PW_TYPE_STRING_PTR,
          offsetof(rlm_eap_ttls_t, default_eap_type_name), NULL, "md5" },
```

```
        { "copy_request_to_tunnel", PW_TYPE_BOOLEAN,
          offsetof(rlm_eap_ttls_t, copy_request_to_tunnel), NULL, "no" },

        { "use_tunneled_reply", PW_TYPE_BOOLEAN,
          offsetof(rlm_eap_ttls_t, use_tunneled_reply), NULL, "no" },

        { NULL, -1, 0, NULL, NULL }     /* end the list */
};

/*
 *      Detach the module.
 */
static int eapttls_detach(void *arg)
{
        rlm_eap_ttls_t *inst = (rlm_eap_ttls_t *) arg;

        if (inst->default_eap_type_name) free(inst->default_eap_type_name);

        free(inst);

        return 0;
}

/*
 *      Attach the module.
 */
static int eapttls_attach(CONF_SECTION *cs, void **instance)
{
        rlm_eap_ttls_t *inst;

        inst = malloc(sizeof(*inst));
        if (!inst) {
                radlog(L_ERR, "rlm_eap_ttls: out of memory");
                return -1;
        }
        memset(inst, 0, sizeof(*inst));

        /*
         *      Parse the configuration attributes.
         */
        if (cf_section_parse(cs, inst, module_config) < 0) {
                eapttls_detach(inst);
                return -1;
        }

        /*
         *      Convert the name to an integer, to make it easier to
         *      handle.
         */
        inst->default_eap_type = eaptype_name2type(inst->default_eap_type_name);
        if (inst->default_eap_type < 0) {
                radlog(L_ERR, "rlm_eap_ttls: Unknown EAP type %s",
                     inst->default_eap_type_name);
                eapttls_detach(inst);
                return -1;
        }
```

```
        /*
         *      Can't tunnel TLS inside of TLS, we don't like it.
         *
         *      More realistically, we haven't tested it, so we don't
         *      claim it works.
         */
        if ((inst->default_eap_type == PW_EAP_TLS) ||
           (inst->default_eap_type == PW_EAP_TTLS) ||
           (inst->default_eap_type == PW_EAP_PEAP)) {
                radlog(L_ERR, "rlm_eap_ttls: Cannot tunnel EAP-Type/%s inside of TTLS",
                     inst->default_eap_type_name);
                eapttls_detach(inst);
                return -1;
        }

        *instance = inst;
        return 0;
}


/*
 *      Free the TTLS per-session data
 */
static void ttls_free(void *p)
{
        ttls_tunnel_t *t = (ttls_tunnel_t *) p;

        if (!t) return;

        if (t->username) {
                DEBUG2("  TTLS: Freeing handler for user %s",
                     t->username->strvalue);
        }

        pairfree(&t->username);
        pairfree(&t->state);
        free(t);
}


/*
 *      Free the TTLS per-session data
 */
static ttls_tunnel_t *ttls_alloc(rlm_eap_ttls_t *inst)
{
        ttls_tunnel_t *t;

        t = rad_malloc(sizeof(*t));
        memset(t, 0, sizeof(*t));

        t->default_eap_type = inst->default_eap_type;
        t->copy_request_to_tunnel = inst->copy_request_to_tunnel;
        t->use_tunneled_reply = inst->use_tunneled_reply;
        return t;
}
```

```
/*
 *      Do authentication, by letting EAP-TLS do most of the work.
 */
static int eapttls_authenticate(void *arg, EAP_HANDLER *handler)
{
        int rcode, i;
        eaptls_status_t  status;
        rlm_eap_ttls_t *inst = (rlm_eap_ttls_t *) arg;
        tls_session_t *tls_session = (tls_session_t *) handler->opaque;
 VALUE_PAIR *password;
 uint8_t *cha;

        DEBUG2("  rlm_eap_ttls: Authenticate");

        /*
         *      Process TLS layer until done.
         */
        status = eaptls_process(handler);
        DEBUG2("  eaptls_process returned %d\n", status);
        switch (status) {
                /*
                 *      EAP-TLS handshake was successful, tell the
                 *      client to keep talking.
                 *
                 *      If this was EAP-TLS, we would just return
                 *      an EAP-TLS-Success packet here.
                 */
        case EAPTLS_SUCCESS:
                eaptls_request(handler->eap_ds, tls_session);
                return 1;

                /*
                 *      The TLS code is still working on the TLS
                 *      exchange, and it's a valid TLS request.
                 *      do nothing.
                 */
        case EAPTLS_HANDLED:
                return 1;

                /*
                 *      Handshake is done, proceed with decoding tunneled
                 *      data.
                 */
        case EAPTLS_OK:
                break;

                /*
                 *      Anything else: fail.
                 */
        default:
                return 0;
        }

        /*
```

```
         *      Session is established, proceed with decoding
         *      tunneled data.
         */
        DEBUG2("  rlm_eap_ttls: Session established.  Proceeding to decode tunneled
        attributes.");

        /*
         *      We may need TTLS data associated with the session, so
         *      allocate it here, if it wasn't already alloacted.
         */
        if (!tls_session->opaque) {
                tls_session->opaque = ttls_alloc(inst);
                tls_session->free_opaque = ttls_free;
        }

        /*
         *      Process the TTLS portion of the request.
         */

        rcode = eapttls_process(handler, tls_session);

        switch (rcode) {
        case PW_AUTHENTICATION_REJECT:
                eaptls_fail(handler->eap_ds, 0);
                return 0;

                /*
                 *      Access-Challenge, continue tunneled conversation.
                 */
        case PW_ACCESS_CHALLENGE:
                eaptls_request(handler->eap_ds, tls_session);
                return 1;

                /*
                 *      Success: Return MPPE keys.
                 */
        case PW_AUTHENTICATION_ACK:

                eaptls_success(handler->eap_ds, 0);

/***************  MODIFICATION OF THE EAP-TTLS: CHENG'S VERSION *****************/
    #ifndef CHENG
                eaptls_gen_mppe_keys(&handler->request->reply->vps,
                                     tls_session->ssl,
                                     "ttls keying material");
   #else
   password = pairfind(handler->request->config_items, PW_PASSWORD);
   eaptls_cheng_mppe_keys(&handler->request->reply->vps,
                                     tls_session->ssl,
                                     "ttls keying material with inner md5", password);
   #endif

/*********************************************************************************************/
```

```
                return 1;

                /*
                 *      No response packet, MUST be proxying it.
                 *      The main EAP module will take care of discovering
                 *      that the request now has a "proxy" packet, and
                 *      will proxy it, rather than returning an EAP packet.
                 */
        case PW_STATUS_CLIENT:
                rad_assert(handler->request->proxy != NULL);
                return 1;
                break;

        default:
                break;
        }

        /*
         *      Something we don't understand: Reject it.
         */
        eaptls_fail(handler->eap_ds, 0);
        return 0;
}

/*
 *      The module name should be the only globally exported symbol.
 *      That is, everything else should be 'static'.
 */
EAP_TYPE rlm_eap_ttls = {
        "eap_ttls",
        eapttls_attach,                 /* attach */
        /*
         *      Note! There is NO eapttls_initate() function, as the
         *      main EAP module takes care of calling
         *      eaptls_initiate().
         *
         *      This is because TTLS is a protocol on top of TLS, so
         *      before we need to do TTLS, we've got to initiate a TLS
         *      session.
         */
        NULL,                           /* Start the initial request */
        NULL,                           /* authorization */
        eapttls_authenticate,           /* authentication */
        eapttls_detach                  /* detach */
};
```

```
/*
 * mppe_keys.c
 *
 * Version:     $Id: mppe_keys.c,v 1.3 2004/02/26 19:04:31 aland 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.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Copyright 2002  Axis Communications AB
 * Authors: Henrik Eriksson <henriken@axis.com> & Lars Viklund <larsv@axis.com>
 */

#include <openssl/hmac.h>
#include "eap_tls.h"

//#define CHENG
/*
 * Add value pair to reply
 */
static void add_reply(VALUE_PAIR** vp,
                      const char* name, const char* value, int len)
{
        VALUE_PAIR *reply_attr;
        reply_attr = pairmake(name, "", T_OP_EQ);
        if (!reply_attr) {
                DEBUG("rlm_eap_tls: "
                    "add_reply failed to create attribute %s: %s\n",
                    name, librad_errstr);
                return;
        }

        memcpy(reply_attr->strvalue, value, len);
        reply_attr->length = len;
        pairadd(vp, reply_attr);
}

/*
 * TLS PRF from RFC 2246
 */
static void P_hash(const EVP_MD *evp_md,
                   const unsigned char *secret, unsigned int secret_len,
                   const unsigned char *seed,   unsigned int seed_len,
                   unsigned char *out, unsigned int out_len)
{
```

```
        HMAC_CTX ctx_a, ctx_out;
        unsigned char a[HMAC_MAX_MD_CBLOCK];
        unsigned int size;

        HMAC_CTX_init(&ctx_a);
        HMAC_CTX_init(&ctx_out);
        HMAC_Init_ex(&ctx_a, secret, secret_len, evp_md, NULL);
        HMAC_Init_ex(&ctx_out, secret, secret_len, evp_md, NULL);

        size = HMAC_size(&ctx_out);

        /* Calculate A(1) */
        HMAC_Update(&ctx_a, seed, seed_len);
        HMAC_Final(&ctx_a, a, NULL);

        while (1) {
                /* Calculate next part of output */
                HMAC_Update(&ctx_out, a, size);
                HMAC_Update(&ctx_out, seed, seed_len);

                /* Check if last part */
                if (out_len < size) {
                        HMAC_Final(&ctx_out, a, NULL);
                        memcpy(out, a, out_len);
                        break;
                }

                /* Place digest in output buffer */
                HMAC_Final(&ctx_out, out, NULL);
                HMAC_Init_ex(&ctx_out, NULL, 0, NULL, NULL);
                out += size;
                out_len -= size;

                /* Calculate next A(i) */
                HMAC_Init_ex(&ctx_a, NULL, 0, NULL, NULL);
                HMAC_Update(&ctx_a, a, size);
                HMAC_Final(&ctx_a, a, NULL);
        }

        HMAC_CTX_cleanup(&ctx_a);
        HMAC_CTX_cleanup(&ctx_out);
        memset(a, 0, sizeof(a));
}

static void PRF(const unsigned char *secret, unsigned int secret_len,
                const unsigned char *seed,   unsigned int seed_len,
                unsigned char *out, unsigned char *buf, unsigned int out_len)
{
    unsigned int i;
    unsigned int len = (secret_len + 1) / 2;
        const unsigned char *s1 = secret;
        const unsigned char *s2 = secret + (secret_len - len);

        P_hash(EVP_md5(),  s1, len, seed, seed_len, out, out_len);
        P_hash(EVP_sha1(), s2, len, seed, seed_len, buf, out_len);
```

```
        for (i=0; i < out_len; i++) {
                out[i] ^= buf[i];
        }
}

#define EAPTLS_MPPE_KEY_LEN     32

#define EAPTLS_PRF_LABEL "ttls keying material"

/*
 *      Generate keys according to RFC 2716 and add to reply
 */
void eaptls_gen_mppe_keys(VALUE_PAIR **reply_vps, SSL *s,
                          const char *prf_label)
{
        unsigned char out[2*EAPTLS_MPPE_KEY_LEN], buf[2*EAPTLS_MPPE_KEY_LEN];
        unsigned char seed[64 + 2*SSL3_RANDOM_SIZE];
        unsigned char *p = seed;
        size_t prf_size;

        prf_size = strlen(prf_label);

        memcpy(p, prf_label, prf_size);
        p += prf_size;

        memcpy(p, s->s3->client_random, SSL3_RANDOM_SIZE);
        p += SSL3_RANDOM_SIZE;
        prf_size += SSL3_RANDOM_SIZE;

        memcpy(p, s->s3->server_random, SSL3_RANDOM_SIZE);
        prf_size += SSL3_RANDOM_SIZE;

        PRF(s->session->master_key, s->session->master_key_length,
           seed, prf_size, out, buf, sizeof(out));

        p = out;

        add_reply(reply_vps, "MS-MPPE-Recv-Key", p, EAPTLS_MPPE_KEY_LEN);
        p += EAPTLS_MPPE_KEY_LEN;
        add_reply(reply_vps, "MS-MPPE-Send-Key", p, EAPTLS_MPPE_KEY_LEN);
}

/********************** MODIFICATION: CHENG'S KEY GENERATION FUNCTION **************/
#ifdef CHENG
void eaptls_cheng_mppe_keys(VALUE_PAIR **reply_vps, SSL *s,
                            const char *prf_label, VALUE_PAIR *value)
{
        unsigned char out[2*EAPTLS_MPPE_KEY_LEN], buf[2*EAPTLS_MPPE_KEY_LEN];
        unsigned char seed[64 + 2*SSL3_RANDOM_SIZE];
        unsigned char *p = seed;
        size_t prf_size;

        prf_size = strlen(prf_label);

        memcpy(p, prf_label, prf_size);
        p += prf_size;
```

```
        memcpy(p, s->s3->client_random, SSL3_RANDOM_SIZE);
        p += SSL3_RANDOM_SIZE;
        prf_size += SSL3_RANDOM_SIZE;

        memcpy(p, s->s3->server_random, SSL3_RANDOM_SIZE);
  p += SSL3_RANDOM_SIZE;
  prf_size += SSL3_RANDOM_SIZE;

  memcpy(p, value->strvalue, value->length);
  prf_size += value->length;

        PRF(s->session->master_key, s->session->master_key_length,
           seed, prf_size, out, buf, sizeof(out));

        p = out;

        add_reply(reply_vps, "MS-MPPE-Recv-Key", p, EAPTLS_MPPE_KEY_LEN);
        p += EAPTLS_MPPE_KEY_LEN;
        add_reply(reply_vps, "MS-MPPE-Send-Key", p, EAPTLS_MPPE_KEY_LEN);
}
#endif
/*****************************************************************************************************/

#define EAPTLS_PRF_CHALLENGE      "ttls challenge"

/*
 *      Generate the TTLS challenge
 *
 *      It's in the TLS module simply because it's only a few lines
 *      of code, and it needs access to the TLS PRF functions.
 */
void eapttls_gen_challenge(SSL *s, char *buffer, int size)
{
        unsigned char out[32], buf[32];
        unsigned char seed[sizeof(EAPTLS_PRF_CHALLENGE)-1 + 2*SSL3_RANDOM_SIZE];
        unsigned char *p = seed;

        memcpy(p, EAPTLS_PRF_CHALLENGE, sizeof(EAPTLS_PRF_CHALLENGE)-1);
        p += sizeof(EAPTLS_PRF_CHALLENGE)-1;
        memcpy(p, s->s3->client_random, SSL3_RANDOM_SIZE);
        p += SSL3_RANDOM_SIZE;
        memcpy(p, s->s3->server_random, SSL3_RANDOM_SIZE);

        PRF(s->session->master_key, s->session->master_key_length,
           seed, sizeof(seed), out, buf, sizeof(out));

        memcpy(buffer, out, size);
}
```

REFERENCES

IEEE Standard

[01] Institute of Electrical and Electronics Engineers, "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", IEEE IEEE Standard 802.11-1997, 1997.

[02] Institute of Electrical and Electronics Engineers, "Local and Metropolitan Area Networks: Port-Based Network Access Control", IEEE Standard 802.1X-2001, June 2002.

[03] Institute of Electrical and Electronics Engineers, "Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks- Specific requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements", IEEE Standard 802.11i-2004, July 2004.

Internet Engineering Task Force(IETF), Request for Comments/Internet Drafts

[04] Ashwin Palekar, Dan Simon, Joe Salowey, Hao Zhou, Glen Zorn, S. Josefsson, "Protected EAP Protocol (PEAP) Version 2", IETF Internet Draft, October 15, 2004

[05] B. Aboba, D. Simon , "PPP EAP TLS Authentication Protocol", RFC 2716, October 1999

[06] B. Aboba, D. Simon, J. Arkko, J. Arkko, H. Levkowetz, Ed., "EAP Key Management Framework", IETF Internet Draft, November 3, 2003

[07] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004.

[08] B. Aboba, P. Calhoun, "RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP)", RFC 3579, September 2003.

[09] C. Adams, S. Farrell, "Internet X.509 Public Key Infrastructure Certificate Management Protocols", RFC 2510, March 1999

[10] C. de Laat, G. Gross, L. Gommans, J. Vollbrecht, D. Spence, "Generic AAA Architecture", RFC 2903, August 2000

[11] C. Rigney, S. Willens, A. Rubens, W. Simpson, "Remote Authentication Dial In User Service (RADIUS)", RFC 2865, June 2000.

[12] D. Whiting, R. Housley, N. Ferguson, "Counter with CBC-MAC", RFC 3610, September 2003

[13] G. Pall, G. Zorn, "Microsoft Point-To-Point Encryption (MPPE) Protocol", RFC 3078, March 2001

[14] G. Zorn, "Microsoft Vendor-specific RADIUS Attributes", RFC 2548, March 1999

[15] G. Zorn, "Microsoft PPP CHAP Extensions, Version 2", RFC 2759, January 2000

[16] G. Zorn, S. Cobb, "Microsoft PPP CHAP Extensions", RFC 2433, October 1998

[17] J. Puthenkulam, "The Compound Authentication Binding Problem", IETF Internet Draft, July 2003

[18] L. Blunk, J. Vollbrencht, "PPP Extensible Authentication Protocol (EAP)", RFC 2284, March 1998.

[19] P. Congdon, B. Aboba, A. Smith, G. Zorn, J. Roese, "IEEE 802.1X Remote Authentication Dial In User Service (RADIUS) Usage Guidelines", RFC 3580, September 2003

[20] Paul Funk, Simon Blake-Wilson, "EAP Tunneled TLS Authentication Protocol", IETF Internet Draft, July 2004

[21] T. Dierks, C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999

[22] Vivek Kamath/Ashwin Palekar, "Microsoft EAP CHAP Extensions", IETF Internet Draft, April 5, 2004

[23] W. Simpson, Ed., "The Point-to-Point Protocol (PPP)", STD 0051 RFC 1661, July 1994.

[24] W. Simpson, "PPP Challenge Handshake Authentication Protocol (CHAP)", RFC 1994, August 1996.

Other References

[25] Arunesh Mishra/William A. Arbaugh, "An Initial Analysis of the IEEE 802.1X Standard", University of Maryland, February 6, 2002

[26] Cameron Macnally, "Cisco LEAP protocol Description", September 6, 2001

[27] Ferguson, N., "Michael: an improved MIC for 802.11 WEP", IEEE 802.11 doc 02-020r0, January 17, 2002.

[28] Housely, R., and D. Whiting, "Temporal Key Hash", IEEE 802.11 doc 01-550r1 October 31, 2001.

[29] Housely, R., /D. Whiting and Ferguson, N., "Alternate Temporal Key Hash", IEEE 802.11 doc 02-282r0 April 2, 2002

[29] Joshua Hill, "An Analysis of the RADIUS Authentication Protocol", InfoGard Laboratories, 2001

[30] Joshua Wright, "ASLEAP: As in "asleap behind the wheel"", http://asleap.sourceforge.net/, 2001

[31] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", FIPS 197. November 26, 2001.

[32] N. Asokan, Valtteri Niemi, Kaisa Nyberg, "Man-in-the-Middle in Tunneled Authentication Protocols", Draft version 1.3, http://eprint.iacr.org/2002/163.pdf, Nokia Research Center, Finland, November 11, 2002

[33] Tom Karygiannis/Les Owens, "Wireless Network Security: 802.11, Bluetooth and Handheld Devices", National Institution of Standards and Technology, November 2002

[34] Vebjorn Moen/H˚avard Raddum/Kjell J. Hole, "Weaknesses in the Temporal Key Hash of WPA", *Mobile Computing and Communications Review, Volume 8, Number 2*, Department of Informatics, University of Bergen.

[35] Wi-Fi Alliance, "Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks", April 29, 2003

VITA

Yu-ming Cheng was born in Taipei, Taiwan, Republic of China, on April 21, 1977. After completing his high school in I-Lan, Taiwan, R.O.C, he entered National Cheng Kung University, Tainan, Tawan, R.O.C, and received his Bachelor's degree in Chemical Engineering in 1999. He then served in the army as a military police in Xindian Military Prison, Xindian, Taiwan, R.O.C, for two years. In summer 2002, he entered Texas State University–San Marcos and completed his Master's degree in Computer Science in 2004.

Permanent Address: 4th Fl., NO. 15, Lane 230, Jingfu St. Wenshan Chu,
Taipei, Tawan, R.O.C (116)

This thesis was typed by Yu-ming Cheng.