Chapter 1 - Introduction

This chapter describes:

- What is RADIUS?
- What is FreeRADIUS?
- FreeRADIUS benefits
- FreeRADIUS case studies

1.0 What is RADIUS?

RADIUS, which stands for “Remote Authentication Dial In User Service”, is a network protocol - a system that defines rules and conventions for communication between network devices - for remote user authentication and accounting. Commonly used by Internet Service Providers (ISPs), cellular network providers, and corporate and educational networks, the RADIUS protocol serves three primary functions:

- Authenticates users or devices before allowing them access to a network
- Authorizes those users or devices for specific network services
- Accounts for and tracks the usage of those services

For a detailed look at how RADIUS performs these functions, see section 2.2, “The RADIUS Session Process”, on page 11.

1.0.1 History

In 1991, Merit Network, a non-profit internet provider, required a creative way to manage dial-in access to various Points-Of-Presence (POPs) across its network. In response to this need, RADIUS was created by Livingston Enterprises.

At the time RADIUS was created, network access systems were distributed across a wide area and were run by multiple independent organizations. Central administrators wanted to prevent problems with security and scalability, and thus did not want to distribute user names and passwords; instead, they wanted the remote access servers to contact a central server to authorize access to the requested system or service. In response to contact from the remote access server, the central server would return a “success” or “failure” message, and the remote machines would be in charge of enforcing this response for each end user.

The goal of RADIUS was, therefore, to create a central location for user authentication, wherein users from many locations could request network access.

The simplicity, efficiency, and usability of the RADIUS system led to its widespread adoption by network equipment vendors, to the extent that currently, RADIUS is considered an industry standard and is also positioned to become an Internet Engineering Task Force (IETF) standard.

1.0.2 Customers

A wide and varied array of businesses currently utilize the RADIUS protocol for their authentication and accounting needs. Some examples of RADIUS customers are:

- Cellular network providers with millions of users
- Small WISP start-up providing the local neighborhood with Internet connectivity
- Enterprise networks implementing Network Access Control (NAC) using 802.1x to ring fence their network

1.0.3 Benefits

The RADIUS client server protocol contains many technological advantages for customers, including:

- An open and scalable solution
- Broad support by a large vendor base
- Easy modification
- Separation of security and communication processes
- Adaptable to most security systems
- Workable with any communication device that supports RADIUS client protocol

The RADIUS client-server architecture provides an open and scalable solution that is broadly supported by a large vendor base. It can be readily modified to meet a variety of situations. Customers can modify RADIUS-based authentication servers to work with a large number of security systems on the market. RADIUS servers work with any communications device that supports the RADIUS client protocol.

In addition, the flexibility of the RADIUS authentication mechanisms allows an organization to maintain any investment they may have made in an existing security technology: customers can modify the RADIUS server to run with any type of security technology. The flexible authentication mechanisms inherent in the RADIUS server facilitate its integration with existing and legacy systems when required.

Another advantage of the RADIUS architecture is that any component of a security system that supports the RADIUS protocols can derive authentication and authorization from the central RADIUS server. Alternatively, the central server can integrate with a separate authentication mechanism.

The utility of the RADIUS protocol extends beyond those systems that utilize network access devices and terminal servers for network access. RADIUS has been widely accepted by Internet Service Providers (ISPs) to provide Virtual Private Network (VPN) services. In this context, RADIUS technology allows an organization to use ISP infrastructure for communications securely.

The distributive nature of RADIUS effectively separates the security processes (carried out on the authentication server) from the communications processes (implemented by the modem pool or the Network Access Server (NAS)), allowing for a single centralized information store for authorization and authentication information. This centralization can significantly lessen the administrative burden of providing appropriate access control for a large number of remote users. If ensuring high availability is not a priority, then redundancy is not required; this centralization can thus be maximized, since all RADIUS-compatible hardware on a LAN can derive authentication services from a single server.

1.1 What is FreeRADIUS?

FreeRADIUS is the most popular and the most widely deployed open source RADIUS server in the world. It serves as the basis for multiple commercial offerings, and it supplies the authentication, authorization, and accounting (AAA) needs of many Fortune 500 companies and Tier 1 ISPs. It is also widely used by the academic community (i.e., eduroam, the world-wide roaming access service developed for the international research and education community, utilizes FreeRADIUS software).

FreeRADIUS was started in August 1999 by Alan DeKok and Miquel van Smoorenburg. Miquel had previously written the Cistron RADIUS server software, which had been widely adopted when the Livingston server was no longer in service. FreeRADIUS was developed using a modular design, to encourage more active community involvement.

1.2 FreeRADIUS Benefits

FreeRADIUS popularity can be attributed to the multitude of additional benefits it offers, far above and beyond those found in the wide variety of other RADIUS servers. FreeRADIUS is based on a feature-rich, modular, and scalable design protocol, which provides the following benefits and advantages to network administrators:

Features

More authentication types are supported by FreeRADIUS than by any other open source server. For example, FreeRADIUS is the only open source RADIUS server to support Extensible Authentication Protocol (EAP).

FreeRADIUS is also the only open source RADIUS server to support virtual servers. The use of virtual servers means that complex implementations are simplified and ongoing support and maintenance costs for network administrators are greatly reduced; thus, the ability of FreeRADIUS to support virtual servers gives it a huge advantage over the competition.

Modularity

The modular design protocol makes FreeRADIUS easy to understand. The modular interface also simplifies adding or removing modules - for example, if a feature is not needed for a particular configuration, the module is easily removed. Once the module is removed, it does not affect server performance, memory use, or security. This flexibility enables the server to run on platforms ranging from embedded systems to multi-core machines with gigabytes of RAM.

Scalability

A single RADIUS server can easily transition from handling one request every few seconds to handling thousands of requests per second, simply by reconfiguring a few default settings. Many large organizations (those with more than 10 million customers) are dependent on FreeRADIUS for their AAA needs. Often, only a single FreeRADIUS server is required to fill the needs of these large organizations.

While many commercial severs offer different versions of their software to handle different needs, only the latest version of FreeRADIUS is needed to obtain better performance, more realms, more RADIUS clients, and many other features, with no need to purchase additional product licenses.

1.3 FreeRADIUS Case Studies

FreeRADIUS can be incorporated to solve a wide range of user authentication issues. A number of detailed case studies, listed below, demonstrate a few FreeRADIUS-based solutions. The type of clients listed in the case studies include large networking companies, ISPs, telecommunications companies, numerous universities, and small businesses.

These real-world case studies have been provided by Network RADIUS SARL. Led by Alan DeKok, Network RADIUS SARL provides FreeRADIUS system support and development to clients worldwide. Alan DeKok continues to be a major contributor to the RADIUS protocol development. A partial list of clients of Network RADIUS SARL include:

- Cisco
- HP
- Alcatel-Lucent
- Covad
- Clearwire
- Etisalat
- Orange

The experience and expertise retained allow Network RADIUS SARL to assist their clients with a range of networking issues and offer a variety of solutions, as illustrated below.

Customer Example - Structured Query Language (SQL)

Our client had designed a new network access system with custom query schema. Their user administration application was using the customer schema queries. When the client tested uninterrupted RADIUS protocol for scalability, they discovered that their RADIUS performance was one percent of the level they required. We were called in to help.

We determined that the table indices in their schema tables were not maximizing efficiently. So, we modified their table indices and updated the RADIUS configuration to use the new column as a key part of its queries.

The performance improvement was dramatic. The changes we made improved the client's system performance three hundredfold. Even better, the changes to the schema did not affect the client's user administration system.

Customer Example - AAA

Our client had used in-house expertise to create the authentication, authorization and accounting (AAA) policies for their organization. While their policies were correct, they began to experience performance problems as their user base grew. By working together, we found a solution.

By comparing their system requirements to the RADIUS server logs, we determined that the system was overloading the database storing their user information. When we examined the AAA policies they had written, the cause was clear.

To fix the problem, we re-wrote key portions of the clients' policies to add preconditions to the database queries. As a result, queries were done only when absolutely necessary. Reducing the number of access request queries was done without affecting any other policies on the system.

When we deployed the new policies, the load on their database dropped by a factor of four hundred. The decrease in load allowed the customer to keep their current systems and eliminated the need for an expensive hardware upgrade.

In addition to using their Network RADIUS SARL support contract to find solutions to problems, the customer was able to save money and optimize operations efficiency.

Customer Example - 802.1X

A customer wanted to deploy 802.1X authentication in his environment to enhance the security of his network. After weeks of effort, they had made no progress. We were called in to help and identified a number of problems.

The customer's networking equipment had a number of firmware issues. We worked with the vendor to fix those and additional issues that resulted from the firmware changes. Using the expertise gained from experience with other customers, we improved our customer's solution to prevent potential future problems.

We worked with an additional equipment vendor to identify undocumented product features to simplify the network configuration. We also tracked down and fixed interaction effects between our product and newer versions of Active Directory that had caused intermittent failures.

The end result was that the customer was able to deploy 802.1X authentication that met all of their requirements. A complex set of features were deployed across a wide range of networking equipment, supplied by a number of different vendors.

The additional network security was unobtrusive to network users. They could continue to use the network in their usual manner. The benefits of authenticating each user on the network were enormous to the network administrator.

Customer Example - Proxy

Our client's legacy system used multiple RADIUS servers to proxy requests to different server destinations. Each RADIUS server implemented a set of policies, and was configured on all of the home servers. This duplication of information increased our customers network maintenance costs.

Upgrading to the FreeRADIUS product meant that the client could choose to continue to use multiple RADIUS servers, or replace them with one server, and update their configuration. If they chose to retain the multiple RADIUS servers, FreeRADIUS could re-use the common configuration. This re-use would lower their ongoing costs.

The client chose to replace the multiple RADIUS servers with a single server. The single server implemented all of their policies in walled garden sites that could not affect each other. The single server also performed all proxying to all home servers.

One result was that proxying became more robust. Instead of each server making fail-over decisions independently, the single server allowed information sharing across each walled garden. This resulted in faster fail-over when there was a problem, and faster fail-back when the home servers returned to service.

Fewer problems for users logging into the network meant increased customer satisfaction. The improved service and reduced maintenance costs also meant fewer support calls. With their end users happy, our customer was happy. Upgrading their legacy systems increased their revenue, and their profit.

Customer Example - Performance

Our customer had installed a basic RADIUS server with an SQL database back end. After a few months of operation, the system was performing at an unacceptable level.

We redesigned their system to allow the RADIUS servers to use the database more efficiently. We added tables, indices, and updated the RADIUS SQL queries. Our improvements helped our customer regain the system performance that they originally had. In addition, the performance gains were maintained even with ten times as much data in the SQL database as before.

Customer Example - Architecture

Our customer had installed a basic RADIUS server. They soon discovered that the performance did not meet their needs. The authentication process request for access response time was not sufficient. Our customer's system accepted load was only a few authentications per second. On a high end machine, this was not acceptable.

We investigated, and determined that the problem was an external network dependency. The RADIUS server was waiting for another network element to respond before returning a response to the Network Access Server (NAS).

After optimizing their network configuration, their network achieved a rate of over 900 authentications per second. This performance level allowed them to move their new system into production.

Chapter Two - How RADIUS works

This chapter describes:

- AAA (Authorization, Authentication, and Accounting)
- RADIUS system components
- RADIUS session process
- RADIUS session messages

2.0 What is AAA?

AAA stands for “Authenication, Authorization, and Accounting”. It defines an architecture that authenticates and grants authorization to users and accounts for their activity. When AAA is not used, the architecture is described as "open", where anyone can gain access and do anything, without any tracking.

It is possible to incorporate only a portion of AAA in a system. For example, if a company is not concerned about billing users for their network usage, they may decide to both authenticate and authorize users, but ignore user activity and not bother with accounting. Similarly, a monitoring system will look for unusual user activity (accounting), but may cede the authentication and authorization decisions to another part of the network.

RADIUS is one of a number of Authentication, Authorization, and Accounting protocols. Other examples of AAA protocols include TACACS+ and Diameter.

Without AAA, a network administrator would have to statically configure a network. Even in the earliest days of dialup access, network administrators found a static model inadequate. AAA ensures the flexibility of network policies. AAA also gives network administrators the ability to move systems; without AAA, they would have to clearly define connectivity options.

Today, the proliferation of mobile devices, diverse network consumers, and varied network access methods combine to create an environment that places greater demands on AAA. AAA has a part to play in almost all the ways we access a network: wireless hotspots use AAA for security; partitioned networks require AAA to enforce access; all forms of remote access use AAA to authorize remote users.

2.1 AAA Definitions

The following sections describe each part of the AAA solution and how each one works.

2.1.1 Authentication

Authentication refers to the process of validating the identity of the user by matching the credentials supplied by the user (for example, name, password) to those configured on the AAA server (for example, name, password). If the credentials match, the user is authenticated and gains access to the network. If the credentials do not match, authentication fails, and network access is denied.

Authentication can also fail if user credentials are entered incorrectly. For example, site policy may allow a user network access from an on-site location using a cleartext password. However, if the same password is entered by the user from a remote location, access may be denied.

An ISP can also choose to deny network access to authenticated users if the users’ account has been suspended. An administrator can choose to permit limited network access to unknown users, as well. For

example, an administrator can provide access to an area where the user can purchase additional network connectivity. This last example is most often seen in for-pay WiFi hotspots.

2.1.2 Authorization

Authorization refers to the process of determining what permissions are granted to the user. For example, the user may or may not be permitted certain kinds of network access or allowed to issue certain commands.

The NAS sends a “request” - a packet of information about the user - and the RADIUS server either grants or denies authorization based solely on information in the “request” sent by the NAS.In each case, the RADIUS server manages the authorization policy and the NAS enforces the policy.

The NAS “request” is really a set of statements. For example, the NAS may send the RADIUS server a “request” containing the following user information:

“user name is Bob”

“password is Hello”

“ip address is 192.02.34”

Once the server receives the request, it uses that information to figure out what properties the user should have (i.e. “Bob” is saying he/she has IP address 192.0.2.34, do the server records contradict this statement?).

The server then sends a reply to the NAS. The reply contains a series of statements about what properties the user should have:

"user name is Bob"

"ip address is 192.0.2.78"

Note that the radius server can’t request further information from the NAS. In contrast with SQL systems, RADIUS is limited in that it cannot make complicated queries. In SQL, queries such as "SELECT name from table where ipaddress = 192.02.34" are common. RADIUS does not have that capability. Instead, RADIUS only makes statements about what is, and what should be.

Upon receipt of a reply from the server, the NAS tries to enforce those properties on the user. If the properties cannot be enforced, the NAS closes the connection.

Authentication vs. Authorization

The following analogy illustrates the difference between Authentication and Authorization:

Imagine you are driving a car and you are stopped by a police officer. The officer asks you to provide a piece of identification to identify yourself. You could, for example, use your passport, driver’s license, or ID card to prove - or *authenticate* - who you are. In terms of the RADIUS protocol, the authentication process identifies the user as someone who is allowed to access the network.

The police officer may also ask you to prove that you are authorized to drive. In this case, there is only one document - a driver’s license - that proves that you are permitted - or *authorized* - to drive a car.

The authorization process thus combines the policy on the RADIUS server and the information in the request from the NAS. The NAS may add additional information to the request, such as the user’s Media Access Control (MAC) address. The NAS sends the information to the server, where an authorization decision is made.

Once the server processes this information, it sends a response to the NAS with instructions detailing which actions are allowed or denied. The NAS then monitors the user's behavior and allows or denies activities according to the policy definition sent by the server.

During the user's network session the policy definition is essentially static. There is no way for the user to request policy changes in RADIUS.

2.1.3 Accounting

Accounting refers to the recording of resources a user consumes during the time they are on the network. The information gathered can include the amount of system time used, the amount of data sent, or the quantity of data received by the user during a session.

During a network session, the NAS periodically sends an accounting of user activity to the server. This accounting is a summary, rather than a complete copy of all traffic. This data is used for billing purposes.

ISPs are a large consumer of accounting data, because each user is billed for every minute of network access. However, corporations have not, historically, relied on network accounting information gathered by RADIUS because employees were not traditionally billed for network access. As their need for ongoing network monitoring increases, though, so does the need to store and process accounting information.

The accounting summary sent by the NAS to the server does not include detailed information such as web sites visited or even how many bytes were transferred using a particular protocol (SMTP, HTTP, and so forth). That type of detailed information is only available to the NAS, and it does not send that data to the server.

If detailed information about user activity is required, network administrators can obtain it through other protocols such as sFlow or NetFlow. However, those protocols are not integrated into RADIUS systems. Network administrators often find it difficult to tie the pieces together to get a more comprehensive understanding of user activity.

2.1.4 Auditing

Auditing refers to the proactive analysis of accounting logs and other data (such as sFlow or NetFlow data). This analysis is a long-term process and is part of ongoing maintenance and monitoring. Auditing provides information about the user's post-authentication behavior. It can provide insight on when to update local site policy to best match user behavior.

Auditing can also be used to determine when an NAS has been compromised, by monitoring NAS enforcement of the required authorization policies. For example, if a user manages to override site policy and log into a particular server when the intent of the site policy was to deny that user access, an audit of the AAA records would highlight that policy violation. Since the intent of the site policy - to deny that user access - was overturned by the user, the audit would indicate that the site policy should be updated by the network administrator to prevent future policy violations. Subsequent audits would monitor long-term behavior and thus ensure that the policy is being enforced.

2.2 RADIUS System Components

RADIUS is a network protocol, a system that defines rules and conventions for communication between network device. Like many protocols, RADIUS uses a client-server model. A RADIUS client (also called a Network Access Server, or NAS) sends requests to a RADIUS server. The RADIUS server then processes the request and sends back a response.

Common NAS products include wireless access points such as the Linksys WRT54G and dial-up equipment commonly available from large network manufacturers. Common RADIUS server products

include Cisco ACS, Microsoft IAS, Funk (now Juniper) Steel Belted RADIUS, Open Systems Radiator, and FreeRADIUS.

While the RADIUS protocol shares the general concept of client-server communication with many other protocols such as HTTP and SMTP, the specifics of RADIUS communications differ. This section describes the RADIUS system in more detail, including the specific roles of the NAS, the server, and databases such as MySQL and Lightweight Directory Access Protocol (LDAP).

See Table 2.1 for a list of RADIUS components and their descriptions.

RADIUS Components		
Component Name	**Functions**	**Examples**
User / Device	Requests access to the network.	Laptop Asymmetric Digital Subscriber Line (ADSL) Modem VOIP Phone
Network Access Server (NAS)	Provides access to the network for the user/device.	Switch Wireless Access Point DSLAM VPN Terminator
Authentication Server	Receives authentication requests from the NAS. Returns authentication results to the NAS. Optionally requests user and configuration information from the database or directory. May return configuration parameters to the NAS. Receives accounting information from the NAS.	FreeRADIUS Radiator IAS NPS ACS
Data Store	Optional database or directory with user authentication and authorisation information. RADIUS server communicates with the data store using DB API or LDAP.	SQL Database Kerberos Service Server LDAP Directory

Table 2.1 RADIUS Components

2.2.1 Network Access Server

The Network Access Server (NAS) acts as the gateway between the user and the wider network. When a user tries to obtain network access, the NAS passes authentication information (for example, user name and password) between the user and the RADIUS server. This process is termed an *Authentication Session*. Note that the user login initiates this *Authentication Session* conversation. This is a key concept.

At the end of the *Authentication Session*, the server instructs the NAS to either reject the user and deny network access or accept the user and provide network access. Once the user has accessed the network, security restrictions (defined by the RADIUS server) are enforced by the NAS, which acts as the gateway router and firewall for that user.

The RADIUS server receives a summary of the user's activities from the NAS. This summary includes data such as session identification information, total time on the network, and total traffic to and from the user. Note that user traffic does not pass through the RADIUS server - the RADIUS server only has access to user information via the NAS summary.

There are many different types of Network Access Servers (NAS). In an enterprise environment, network switches and wireless access points act as NASs to ensure only authorized users may access the corporate network. In contrast, carriers may use ADSL terminators or Digital Subscriber Line Access Multiplexers (DSLAM) as NASs to authenticate users and generate accounting information for billing. In fact, any device or application that verifies username and password authentication may be a RADIUS client.

RADIUS client NASs include FTP servers, web servers, and Unix login services.

Using the term server in reference to the Network Access Server can create confusion, because the NAS acts as a client in the RADIUS protocol. This document uses the term NAS to refer to a client and the term server to refer to a RADIUS Server.

The NAS starts all RADIUS conversations (Authentication Sessions) when a user requests network access.

2.2.2 RADIUS Server

The RADIUS server is usually a software application running on a Blade or self-contained server. RADIUS appliances with simplified maintenance and management interfaces are also available. In either case, the function of the server is identical: the server waits for a request from an NAS, processes or forwards the request, and then returns a response to the NAS. The response can contain authorization policies or an acknowledgment of accounting data received.

A single RADIUS server can receive and process many simultaneous access requests from numerous types of NASs (such as ADSL, dial-up, or VPN concentrators) in many different locations. A single server may also interact with flat files, SQL databases, LDAP directories, or other RADIUS servers. In order to make a decision regarding an access request, the RADIUS server must first use information from many sources.

Once the server makes a decision, it returns a response to the NAS. The NAS may enforce the policy in that response, or it may ignore it altogether. The server has no way of knowing if the NAS has received its response, or if the NAS is obeying the instructions in that response. Since it is customary for the NAS to log very little information about what has been received or how server responses are processed, it is very difficult to create and debug local site policies.

The RADIUS server has no control over what the NAS does with a response.

Consider the following analogy to help illustrate the point: a Human Resources (HR) department acts like a RADIUS server, by setting policies, and a security guard acts like the NAS in a network, by carrying out those HR department policies.

In this example, the company policy is that when an employee is fired, HR notifies security and removes building access from that employee. The security guard is then responsible for ensuring the fired employee no longer accesses the company building. If one day an employee gets fired (similar to a user being denied access) and the HR department informs the security guard that the employee is no longer

free to come and go (similar to the RADIUS server decision sent to the NAS), it is then up to the security guard at the company front desk to perform the task of refusing entry to the fired employee (similar to the NAS enforcing system access in a network). In the network, the NAS enforces system access. The RADIUS server does little more than offer advice to the NAS.

RADIUS Server Policies

The RADIUS server processes an NAS request based on the following criteria:

- Contents of the NAS request
- Information available locally to the RADUS server (flat files, SQL, LDAP)

The limitations inherent in the above processing criteria mean that the server cannot negotiate with an NAS to request more information: the server simply takes what the NAS sends and returns either an acknowledgment or a non-acknowledgment. This limitation is another key concept.

The RADIUS server has no control over the content of the request that the NAS sends.

Thus, once the RADIUS server receives the request from the NAS, it must use local information - policies or rules that a network administrator created and configured within the server - to decide how best to respond to the NAS request. The policies may be simple, such as “accept anyone with a correct user name and password”. Or, they may be complicated, such as “allow basic users to request premium services in non-premium hours, except for Sundays and holidays, so long as their payment status is up to date”.

In all cases, the policies must be designed, implemented, and deployed by the network administrator; this can be a significant effort, because policies are based on the contents of the NAS requests. Note that the NAS documentation does not always describe the content of the NAS requests; thus, in most cases, the only way for a network administrator to determine the NAS request content is to set up a test network. Test logins will result in the receipt of requests by the server. The administrator can then examine these requests to determine their content and create policies that look for those specific sets of attributes; once the policy is created, the server then uses that information to make decisions.

This process becomes more complicated when different NAS elements send the same information in different formats. For example, RADIUS has no MAC address data type, which means that the MAC address is sent as ASCII strings. Some NAS elements send a MAC address in the format of “00:01:02:03:04:05”, while others use the format “00-01-02-03-04-05”. The fact that these differences are not documented makes policy creation very difficult.In most cases, the administrator has to resort to trial and error methods to determine how to implement policies.

2.2.3 Data Stores

Data stores (i.e., databases or directories) permit the storage and retrieval of data. They have limited decision-making capabilities. While stored procedures are possible in most databases, they are rarely used when simple data storage is required.

The key differences between RADIUS servers and data stores are the way they support policies and authentication. The role of a data store in the authentication process is to provide data to a RADIUS server. The server then uses an authentication method to authenticate the user.

When a RADIUS server authenticates a user or stores accounting data for that user, it reads from or writes to a database or directory.User information (i.e., user name, password, credit amount) and session data (i.e., total session time and statistics for total traffic to and from the user) are stored on this database or directory.

In many respects, the RADIUS protocol is similar to a remote database query language. Specifically, while an SQL or LDAP database stores user data, that database cannot be queried directly by the NAS. Instead,

the NAS sends a request to the server, which in turn queries the database. This simplification of the normal database query language means that it is easy to add authentication and accounting to an NAS instead of implementing a full-featured SQL client, which would be very resource intensive and costly.

Key Differences Between RADIUS Servers and Data Stores	
RADIUS Servers	**Data Stores**
Implement policies	Rarely implement policies
Support complete authentication protocols sets, such as: • CHAP • MS-CHAP • MS-CHAPv2 • 802.1X (EAP, EAP-TLS, PEAP, EAP-TTLS, EAP-MD5, EAP-GTC, LEAP) • HTTP Digest authentication	Permit simple authentication queries, such as: • LDAP "bind as user"

Table 2.1.3 Key Differences Between RADIUS Servers and Data Stores

Data stores store data. They do not authenticate users.

2.2.4 RADIUS System Components Summary

In summary:

- An NAS is responsible for requesting and enforcing network access, filtering traffic, and sending summaries of accounting data.
- The RADIUS server is responsible for receiving access requests, interpreting complex policies, and returning a response to the NAS.
- A data store (i.e., directory or database) is responsible for storing large amounts of data, most often keyed by user name. This data may include user passwords, credit amounts, session data, and more.

2.3 The RADIUS Session Process

A RADIUS session consists of the following steps:

1. A remote user connects to a RADIUS client device (using Point-to-Point Protocol (PPP, 802.1X) or another Data Layer link protocol) and initiates a login:
    - The NAS initiates all conversations (*Authentication Sessions*) in RADIUS.
    - All information sent to the server is done solely at the discretion of the client.
    - The RADIUS server does not control what the NAS sends.

2. The Network Access Server communicates with the RADIUS server using a shared secret mechanism:
   - RADIUS uses User Datagram Protocol (UDP) port 1812 for authentication and 1813 for accounting.
3. The NAS sends a RADIUS message (called an *Access-Request*) to the server.
   - This message contains information about the user, including user name, authentication credentials, and requested service.
   - In addition, the message may contain information about the NAS, such as its host name, MAC address, or wireless SSID.
   - The message is sent using the Password Authentication Protocol (PAP), Challenge-Handshake Authentication Protocol (CHAP), or Extensible Authentication Protocol (EAP).
   - The server must decide whether to authenticate or authorize a user based solely on the information contained within the NAS request, as it does not have access to any additional user information.
   - If the NAS sends a packet with an authentication protocol that the server does not support, the server will reject that request.
4. The RADIUS server processes the request and verifies the login request against either a local database or the authentication service running on the network.
   - Authentication services can include: LDAP servers for a domain validation; Active Directory servers on Windows networks; Kerberos servers; SQL Server or another type of database for getting information from a database
5. The RADIUS server sends validation results back to the NAS in one of the following forms: *Access Reject*, *Access Challenge*, or *Access Accept*.
   - *Access Reject* locks the user out of the network if the user is invalid or not authorized, denying them access to the requested resource.
   - *Access Challenge* occurs when the server requires additional information from the user. Since RADIUS packages are limited in size, *Access Challenges* allow the exchange of larger amounts of data.
   - *Access Accept* provides the user access to the resource and contains policy information which the NAS uses to provide services to, and enforce the behavior of the user. An *Access Accept* condition does not apply to all resources. Each additional resource is checked as required. The RADIUS client also verifies the original access offered on a periodic basis.
   - An *Access Accept* response results in the NAS providing the following services to the remote client: supplying a static or dynamic IP address; assigning a Time-to-Live for the session; downloading and applying the users' Access Control List (ACL); setting up any L2TP, VLAN, and QoS session parameters required

6. Once the session is established on the RADIUS client, the accounting process is initiated:
   - The `Accounting-Request (start)` message, sent by the NAS to the server, indicates the commencement of the session. The the session account record is then created.
   - The `Accounting-Request (stop)` message indicates the end of the session; the session account record is then closed.
   - The data stored in the database during the accounting sessions is used to generate billable information and reports.
   - Accounting information retained in the database includes the following: time of session, number of packets and amount of data transferred, user and machine identification, network address, and point of attachment information.

2.4 RADIUS Session Messages

A RADIUS session message consists of a single User Datagram Protocol (UDP) packet, containing a short header followed by the authentication, authorization, or accounting data.

2.4.1 Message Attributes

Each message contains a list of Attribute Value Pairs (AVPs), commonly referred to as *attributes*. These *attributes* carry information from the NAS to the server or virtual proxy server and from the server to the NAS. Common attributes include items such as user name, password, IP address, and NAS address; each attribute contains only one of these items. An attribute can also contain sub-attributes, for grouping purposes.

For a complete list of standardized attributes, see the FreeRADIUS RFC Attributes page on **http://freeradius.org/rfc/attributes.html**.

Each client and server supports only a limited set of attributes. In some cases, attributes may not be supported because the server or NAS software may have been written before a standard was published, or the software may simply not support that particular functionality. The best way to know which attributes are supported is to consult the software vendor documentation. If the documentation is silent on the topic, the attributes in question are probably not supported. Contact the vendor for additional information.

In addition to standardized attributes, vendors can extend RADIUS with vendor-specific attributes (VSAs). Using VSAs means that the vendor can rapidly add functionality without having to do a time-consuming standardization process.

In order to be useful in the RADIUS client protocol, however, VSAs must be defined on the RADIUS server. Because VSAs are non-standardized attributes, it is difficult to discover any information about them. In addition, they are all vendor-specific (only work on that vendor's products). Thus, defining VSAs for RADIUS server use may be difficult.

2.4.2 Attribute Types

Basic RADIUS attribute types are defined in RFC 2865 (http://tools.ietf.org/html/rfc2865). Since the original implementation and standardization, additional attribute types have been also defined. Table 2.3.2 lists attribute types and their formats.

RADIUS Attribute Types		
Type	**Format**	**Notes**
`integer`	unsigned 32-bit integer	
`ipaddr`	IPv4 address	
`date`	Unix timestamp in seconds since January 1, 1970 GMT	
`string`	Variable-length string field	Used by FreeRADIUS for printable text strings.
`octet`	Variable-length string field	Used by FreeRADIUS for binary data.
`ifid`	IPv6 interface ID	
`ipv6addr`	IPv6 address	
`ipv6prefix`	IPv6prefix	
`byte`	Single-byte integer	Added in FreeRADIUS v2.0. Values between 0 - 255.
`short`	Two-byte integer	Added in FreeRADIUS v2.0. Values between 0 - 65535 Added in FreeRADIUS v3.0 integer64, 8-byte integer

Table 2.3.2 RADIUS Attribute Types

2.4.3 Attribute Definition Dictionaries

FreeRADIUS contains over 100 dictionaries, which hold nearly 5000 attribute definitions.

Unlike text-based protocols such as SMTP or HTTP, RADIUS is a binary protocol; therefore, although attributes are commonly referred to by name (for example, “User-Name”), these names have no meaning in the protocol. Data (i.e., “User-Name”) are encoded in a message as a binary header with binary data, and not as text strings.

Dictionary files are used to map between the names used by people and the binary data in the RADIUS packets.

The packets sent by the NAS contain attributes that have a number, a length, and binary data. By contrast, a dictionary file consists of a list of entries with name, number, and data type.

The server uses the dictionaries to interpret the binary data as follows: the server searches the dictionaries to match the number from the packet, and the corresponding data type in the dictionary entry is then

used by the server to interpret the binary data in the packet. The name within the dictionary entry appears in all logs and debug messages, in order to present the attribute in a form that is understandable to people.

This process also works in reverse: the server uses the dictionaries to encode a string (i.e., "User-Name = Bob") as "number, length, binary data" in a packet.

For example, a server may receive a packet that contains the number 1 and some binary data. The server would not be able to decode these attributes without knowing if the binary data were in the form of a string, an IP address, or an integer. The corresponding dictionary entry may state: the number '1' is called 'User-Name' and it is in the form 'string'. This information (i.e., form = 'string'), contained in the dictionary entry, allows the server to successfully decode the packet.

Adding new attributes to RADIUS software is made simpler through the use of dictionaries, as they allow the administrator to create new mappings without upgrading the software.

Vendors can define new attributes in the dictionary without changing any of the server or client source code. These new attributes can then be incorporated in policy decisions or logged in an accounting record.Vendors can also define new functionality for their equipment by publishing a new dictionary file. Server support for an NAS can easily be added by writing the correct dictionary file for that NAS.

Note that there is no direct connection between the NAS and the dictionary files, as the dictionary files reside only on the server. For example, if names are edited in a local dictionary file, there is no effect on any other NAS or RADIUS server, because these names appear only in the local dictionary file. The RADIUS packets remain in binary format and contain only numbers, not names.

Editing attribute names in a local dictionary file does not effect any other NAS or RADIUS server. Only the attribute numbers are encoded and sent in RADIUS packets.

NOTE that the NAS and server may use different dictionaries, which may cause problems if they are not coordinated (a set of data may be interpreted differently by the NAS and the server, i.e., as a string by the NAS and an IP address by the server)

Since the attribute names are not sent in RADIUS packets, the dictionaries are limited to the local server and to the policy implemented by that server.

In this respect, NAS implementations are much simpler than server implementations. Each NAS "looks" in the RADIUS packet for what it "understands" and ignores everything else. In contrast, each RADIUS server is presented with all of the information from every NAS in the RADIUS deployment. Each RADIUS server must be capable of "understanding" the functionality and configure-ability of every attribute that is necessary to authenticate or authorize the users.

2.4.4 Dictionary File Format

It is very important to use the correct dictionaries. If the wrong dictionaries are used, the server may not properly interpret local configuration, or generate the correct response for the NAS.The format of the RADIUS dictionary files is not currently standardized, although most are simple variants of the original Livingston format defined in 1992. Because of the differences in format, a server's dictionary files may be incompatible with different versions of the same RADIUS server software; one server's dictionary files can also be incompatible with another server's dictionary files.

If the server does not have access to the correct vendor-specific attributes and dictionaries, it cannot correctly process any VSAs it receives for that vendor.

Servers require access to a vendor dictionary to understand vendor attributes.

Some servers may not contain all the necessary dictionary files because each vendor defines their own dictionaries and chooses when to inform the server vendor of those changes. Thus, if a server does not include dictionaries for a particular vendor, contact that vendor, and not the organization that supplied the RADIUS server.

2.4.5 Server-side Attributes

Server-side attributes are attributes that control the server's behavior. It is frequently necessary to define these server-side attributes, while ensuring that information pertaining to server-side attributes never gets sent through the network in a RADIUS message. Server-side attributes should not be included in RADIUS messages, since these attributes are internal to server implementation.

Definitions for server-side attributes may vary by server vendor, or may vary even from one version of the same server to another. Only FreeRADIUS definitions for internal attributes are referenced in this document. Those definitions are generally the same across all versions of the server, but other vendors may have different implementations.

Information such as "use LDAP server X", or "remember that the user is in group Y" should be used to create local policy. This information should be stored in server-side attributes (also known as "non-protocol attributes").

Server-side attributes are presented using the same format as standard or vendor RADIUS attributes. This format gives the administrator the same control over internal aspects of the server behavior as over the server external responses. The server-side attribute information can be retrieved as part of one policy and checked later as part of another policy. For example, the policy can say "use LDAP server X for this request" and "respond with attribute X, value Y".

2.4.6 Processing Requests

The server processes requests through local site policy. That policy is used to examine the request, the request attributes, and the attribute values. The server then builds a reply message using responses (determined by local policy) such as time of day restrictions, group access limitations, and IP address allocation. The processing stage may include keeping track of server-side attributes, as described in **2.4.5 Server-side Attributes** on page 18.

FreeRADIUS maintains three attribute lists for every request. They are the:

- Request list: the attributes in the request packet
- Reply list: the attributes sent in the reply packet
- Config list: the attributes that control internal server behavior

Not all server implementations behave exactly this way, but they usually have similar functionality.

Chapter Three - Installation Requirements

This chapter describes:

- Installation options
- Installing FreeRADIUS

FreeRADIUS version 2.2.1 is used for all instructions and examples herein.

This chapter also includes recommendations regarding:

- Pre-installation planning of requirements and key processes
- Pre-built packages for installation
- Installation from source

Detailed implementation and installation procedures are available in the FreeRADIUS Implementation Guide.

A FreeRADIUS Reference Guide is also available. The FreeRADIUS Reference Guide provides detailed information on configuration files, as well as help for troubleshooting common installation and configuration issues.

3.0 Installation Options

The easiest way to install FreeRADIUS, is to use one of the available pre-built binary packages. Binary packages exist for Ubuntu or Debian-based systems and for RedHat-based systems.

For non-Linux systems, the FreeRADIUS source code can be compiled and installed; however, this form of installation is not recommended unless access to in-house network administration experts is readily available. Contact Network RADIUS SARL (SALES@NetworkRadius.com) for assistance if this is your installation option. Network RADIUS SARL has expert consultants available to help with both planning and implementation of a FreeRADIUS source code customer install.

3.0.1 Life Cycle of a Server

This section describes how to plan for the full life cycle of a server.

Installation is the first step in the life cycle of a server, and, with installation, the ongoing server processes of testing, maintenance, and auditing begin. Part of the process of installing a server should, therefore, be planning for the full life cycle of the server.

The first step in building a server is designing the surrounding ecosystem. This ecosystem includes the databases needed by the server, the external NASes with which the server will communicate, and the network configuration, among other factors.

Every aspect of the surrounding ecosystem should be documented for long-term maintenance.

Designing complex systems can be difficult, and is outside of the scope of this document. It is important to note that a RADIUS server, which is required to fulfill ongoing operational goals that include more criteria than "responds to one or two packets", cannot just simply be installed and expected to “work”.

The architecture of the ecosystem is often critical for performance and stability. For example, a RADIUS server can easily handle receiving tens of thousands of `Accounting-Request` packets per second. However, if it needs to do anything with those packets (such as write them to a database), the system may only be able to handle a fraction of that amount - perhaps hundreds of packets per second.

Once the system has been designed, the next step is the installation and configuration of the basic services. This process ensures that the servers are running and ready for traffic.

The installation process is covered in more detail below.

After installation, the systems should be configured for ongoing operations. The configuration process involves checking that log files are rotated regularly, old records are cleaned from the accounting tables, regular backups are configured, and monitoring and alerting is in place. The goal is to ensure that the systems are stable and can operate without constant intervention by the administrator.

Following configuration of the systems, testing is the next step.

The testing should be automated and documented. The systems should be tested for performance, compliance to the requirements, stability, and the ability to deal with edge conditions. All monitoring and alerting processes should be tested, to ensure that the alerts are generated as expected. Proper testing of the alert system can ensure that the worst possible outcome - an alert system failing to generate alerts when necessary - can be avoided.

Ideally, there should be a "lab" system, used only for testing, and a "production" system for live traffic. When the system is first built, prior to it being made live, a first set of “acceptance” tests should be run on the production system to ensure it behaves as desired. Once the system is up and running, if any changes are made, these same tests should be run on the lab system to be sure the changes work, and then the changes can be pushed to production. If this process is not followed, then any changes to the production system may cause service outages, with serious consequences for ongoing operations.

All changes should be tracked in a revision control system. This process ensures that reverting to a "known working" configuration is possible if there are problems in the production environment. It also enables the production configuration to be replicated to another system for redundancy and high availability.

When the systems reach their end of life, the new systems should be installed as above, and then the old systems should be decommissioned. We recommend leaving the old systems up and running for some time after the new systems are operational. The cost of leaving the old system running is small when compared with the risk of being unable to revert to a "known working" configuration.

The above description is a short introduction to the life cycle of a radius server. More detailed documentation is available from Network RADIUS SARL.

3.0.2 Dependencies

Any LDAP and SQL dependencies require their respective client libraries to communicate with the LDAP or SQL server. The pre-built FreeRADIUS packages do not include these external dependencies.

Installation from source requires a working C compiler, such as the Gnu Compiler Collection or Clang, and the GNU Make utility. If your system does not have these tools installed, the `configure` process will fail with an error. To correct this problem, install those tools, and re-run `configure`. Pre-built packages are recommended wherever possible.

3.0.3 Downloading FreeRADIUS

Binary Packages

Binary packages for many platforms are available from third party Web sites. A list of links to these third party sites is maintained at **http://freeradius.org/download.html**.

FreeRADIUS Source Code

Installation from the source code is not recommended. If you choose to install from source, we recommend contacting Network RADIUS SARL for assistance.

For any non-Linux system, compiling and installing the FreeRADIUS source code rather than using a pre-built binary package is recommended. Other reasons to install the source code include:

- no pre-built installation packages are available for the system in use
- to install an experimental module
- to make local changes to the source (not recommended)

The most recent FreeRADIUS source code is available from **http://freeradius.org**. To download the source code, follow these instructions:

1. Click the **Download** link to navigate to the Downloads page.
2. Click the **tar.gz** link.
   The tar.gz file starts downloading.
3. Save the file to disk.

3.1 Installing FreeRADIUS

3.1.1 Installing From a Pre-built Binary Package on Ubuntu

Ubuntu systems use the `apt` suite of tools to find and install packages. The following command will find all packages related to FreeRADIUS:

```
$ apt-cache search freeradius
```

The above Ubuntu search returns the following packages:

- `freeradius` - a high-performance configurable RADIUS server
- `freeradius-common` - FreeRADIUS common files
- `freeradius-dbg` - debug symbols for the FreeRADIUS packages
- `libfreeradius-dev` - FreeRADIUS shared library development files
- `libfreeradius2` - FreeRADIUS shared library
- `freeradius-dialupadmin` - set of PHP scripts for administering a FreeRADIUS server
- `freeradius-iodbc` - iODBC module for FreeRADIUS server
- `freeradius-krb5` - kerberos module for FreeRADIUS server
- `freeradius-ldap` - LDAP module for FreeRADIUS server
- `freeradius-mysql` - MySQL module for FreeRADIUS server
- `freeradius-postgresql` - PostgreSQL module for FreeRADIUS server
- `freeradius-utils` - FreeRADIUS client utilities
- `libfreeradius3` - FreeRADIUS shared library

The first package (`freeradius`) is the base server. All the modules necessary for a typical configuration are installed using this package. The other packages are optional modules that add functionality such as LDAP or MySQL.

As many optional packages as desired may be added to the base server. Installing additional packages does not affect operational performance or memory usage of a FreeRADIUS server.

The FreeRADIUS server may also be installed and run as a daemon via the following command:

```
$ sudo apt-get install freeradius
```

If the server is not in operation (production), then the service can be stopped without negative effects at anytime simply by running it in debugging mode and using the following command:

```
$ sudo service freeradius stop
```

The rest of this document assumes that the FreeRADIUS binary is available in the PATH environment variable. The command to run the server is herein referred to as "`radiusd`".

3.1.1.1 Debian-based Systems

Several things may be different with Debian-based systems. Some Debian-based systems call the binary `freeradius` instead of `radiusd`. An alias can be created by performing the following commands:

1. Change the directory to:

```
$ cd /usr/sbin
```

2. Create a symbolic link:

```
$ sudo ln -s freeradius radiusd
```

The rest of this document assumes that the reader is familiar with the location of the various configuration files. In Debian-based systems the configuration files are located in the directory `/etc/freeradius`. *This document refers to the* `raddb` *directory instead of* `/etc/freeradius`.

3.1.1.2 Ubuntu System Firewalls

Recent editions of Ubuntu systems have a firewall feature. To allow RADIUS traffic to reach the server through the firewall, rules must be added. The following rules allow for packets to be sent from any system to the RADIUS server. :

1. Add a firewall rule to allow RADIUS authentication requests on port 1812 by typing the following:

```
$ ufw allow 1812/udp
```

2. Add a firewall rule to allow RADIUS accounting requests on port 1813 by typing the following:

```
$ ufw allow 1813/udp
```

3. To verify that the rules have been installed correctly, run the following command:

```
$ ufw status | grep 181
```

The following confirmation should appear on the monitor:

```
1812/udp           ALLOW       Anywhere
1813/udp           ALLOW       Anywhere
```

The above rules allow for the reception of packets from any system. However, all packets not included on an internal RADIUS server systems list are excluded, and thus the RADIUS server is not compromised. The default firewall setting on the server is to ignore all external packets and only allow packets from `localhost`.

3.1.2 Installing From a Pre-built Binary Package on RedHat

RedHat systems use the `yum` tool to find and install packages. Run the following command to find all packages related to FreeRADIUS:

```
$ yum search freeradius
```

The search finds the following packages. Note that some lists may differ slightly depending on the system:

- **`freeradius`** - FreeRADIUS server.
- **`freeradius-mysql`** - MySQL bindings for FreeRADIUS
- **`freeradius-postgresql`** - postgresql bindings for FreeRADIUS
- **`freeradius-unixODBC`** - unixODBC bindings for FreeRADIUS
- **`freeradius2`** - FreeRADIUS server
- **`freeradius2-krb5`** - Kerberos 5 support for FreeRADIUS
- **`freeradius2-ldap`** - LDAP support for FreeRADIUS
- **`freeradius2-mysql`** - MySQL support for FreeRADIUS
- **`freeradius2-perl`** - Perl support for FreeRADIUS
- **`freeradius2-postgresql`** - Postgresql support forFreeRADIUS
- **`freeradius2-python`** - Python support for FreeRADIUS
- **`freeradius2-unixODBC`** - Unix ODBC support for FreeRADIUS
- **`freeradius2-utils`** - FreeRADIUS utilities

The first package above (`freeradius`) is the base server for version 1. To install version 2, the `freeradius2` package, which installs all the necessary modules for a typical configuration, is recommended.

The other packages are optional modules that add functionality, such as LDAP or MySQL. You can add as many optional packages as desired. Installing packages does not affect the performance or memory usage of an operational FreeRADIUS server.

To install the FreeRADIUS server and run it as a daemon, use the following command:

```
$ sudo yum install freeradius2
```

If the server is not in operation (production), the service may be stopped at any time and without negative effect by running the server in debugging mode and entering the following command:

```
$ sudo service freeradius stop
```

In the rest of the document it is assumed that the FreeRADIUS binary is available in the PATH environment variable. In RedHat-based systems, the binary is called `radiusd`. Only a root user is permitted to enact the commands in the next section.

In the rest of the document it is assumed that user is familiar with the location of the various configuration files. RedHat-based systems put the configuration files into the directory `/etc/raddb` This document refers to the `raddb` directory instead of `/etc/raddb`.

3.1.2.1 RedHat System Firewalls

Recent editions of RedHat systems have a firewall feature. Network Administrators must add the following rules to allow RADIUS traffic to reach the server.

1. Add a firewall rule allowing RADIUS authentication requests on port 1812 by typing the following:

```
$ iptables -A INPUT -p udp --dport 1812 -j ACCEPT
```

2. Add a firewall rule allowing RADIUS accounting requests on port 1813 by typing the following

```
$ iptables -A INPUT -p udp --dport 1813 -j ACCEPT
```

3. To verify that the rules are installed, run the following command:

```
$ iptables -L -n | grep 181
```

The above rules allow for the reception of packets from any system. However, all packets not included on an internal RADIUS server systems list are excluded, and thus the RADIUS server is not compromised. The default firewall setting on the server is to ignore all external packets and only allow packets from `localhost`.

These rules allow any system to send packets to the RADIUS server. Allowing the server to receive packets from any system does not compromise the RADIUS server because the server has a list of systems from which it is permitted to receive packets. The default firewall setting is set to ignore all external packets and only allow packets from `localhost`.

3.1.3 Installing From Source

Although you can install FreeRadius from source, this approach is not recommended for most implementations.

Installing from source requires significant expertise. Unless you have in-house experts, using the pre-built options is recommended. If you don't have in-house experts, but have unique installation requirements that must be supported by a custom installation from source, contact Network Radius SARL for consulting support (SALES@NetworkRadius.com].

Before installing FreeRADIUS, log in as root. The `make install` process installs the server in the `/usr/local/` directory. The following steps will install the FreeRADIUS program:

```
$ tar -zxf freeradius-server-2.2.1.tar.gz
$ cd freeradius-server-2.2.1
$ ./configure
$ make
$ sudo make install
```

To change the location where the server is installed, use this command:

```
$ ./configure --prefix=/opt
```

See `./configure --help` *for a complete list of options available for* `configure`.

When installing from source, if Extensible Authentication Protocol (EAP) for WiFi, WiMAX, or 802.1x is used, test certificates will also need to be created. Test certificates do not need to be created when installing pre-built packages.

3.1.3.1 Modules

The configure process builds all possible modules. It does this by looking at the freeRADIUS code (i.e., rlm_sql) and determining which external libraries that the module needs (e.g., sql client libraries). If the dependency is available, then that module is built.Where a dependency is missing, that module is not built. If some modules are not available at that time, the server will still function. However, any missing modules may affect additional functionality (for example, MySQL may not be available).

Most optional modules, such as MySQL, are installed from client libraries and developmental header files on the user's system. Because of differences in systems, this document cannot cover all possible permutations. Pre-built system-specific packages are recommended whenever possible.

3.1.3.2 Verifying the Installation

To verify the FreeRADIUS install, run the server in debugging mode. If the system installed correctly, the screen will show a large amount of text that ends with the message "Ready to process requests". This message means that the FreeRADIUS server was successfully installed. An example of the verification message is shown below:

```
$ radiusd -X
...
Listening on authentication address * port 1812
Listening on accounting address * port 1813
Listening on authentication address 127.0.0.1 port 18120 as server
inner-tunnel
Listening on proxy address * port 1814
Ready to process requests.
```

The rest of this document assumes that the FreeRADIUS binary is available in the PATH environment variable. Only a root user can access the following procedures.

The rest of this document also assumes that the reader is familiar with the location of the various configuration files. Note that this document refers to the raddb directory, instead of `/usr/local/etc/raddb`.

3.1.3.3 Installation Overview

The installation process is fail-safe. We do not recommend re-installing the server on a production system, although it is possible to do. As well, the build process can be re-run as many times as required without causing any problems; it is always safe to re-run the build process.

The base level of functionality allows the server to perform accounting, along with PAP, CHAP, MS-CHAP, and a few EAP authentication types. Installing the server does not over-write your configuration.

One issue that may arise is absent dependencies after a build.The FreeRADIUS server has been designed to minimize the number of external dependencies; thus, if a third-party library is unavailable, any feature using that library is also unavailable. When the server is built, it includes everything that is available on the local system, e.g. SQL, LDAP, if they exist locally. If it is discovered that required dependencies are absent following a server build, it is thus necessary to install the appropriate third-party libraries on the local system.

If third-party libraries need to be added after completing an install, simply re-run the build process after the new library has been installed on the server. This sequence may be repeated as many times as necessary to complete installation of all third-party libraries.

Chapter 4 - Configuration Requirements

This chapter describes the configuration FreeRADIUS file structure and the purpose of the configuration file types:

- Core configuration files
- The radiusd.conf file
- File format
- The packet processing sections
- The users file

The `/etc/raddb` directory contains the configuration files. The main configuration file, which is named `radiusd.conf`, references the other configuration files. The FreeRadius package contains a large number of files, but many of the configuration files are examples; the rest of the configuration files ensure that the server functions correctly out of the box.

4.0 Core Configuration Files

The configuration files are contained in the `raddb` directory. The configuration file structure also contains sample virtual server configurations. The server includes a number of configuration files not listed in this section. For example, each of the fifty plug-in modules has its own configuration file, none of which are listed below. The list below identifies the most important configuration files, listed in alphabetical order.

- **`radiusd.conf`** Defines the configuration parameters for the RADIUS server. It includes references to all of the other configuration files.
- **`clients.conf`** Defines information necessary to configure the RADIUS client, including IP addresses and shared secrets. This file is referenced from the `radiusd.conf` file.
- **`dictionary`** Defines local attributes for the RADIUS server. This file references the default dictionary files. The default dictionary files include thousands of attribute definitions for over one hundred vendors.
- **`proxy.conf`** Defines upstream home servers, including information on IP addresses and shared secrets. It also defines Realms. The `radiusd.conf` file references the `proxy.conf` file.
- **`sites-enabled/default`** This is the default virtual server. This file handles authentication and accounting requests. It contains a configuration designed to work with the largest number of authentication protocols. The `radiusd.conf` file references the `sites-enable/default` file.
- **`sites-enabled/inner-tunnel`** This virtual server handles authentication methods that are carried inside of a TLS tunnel, as part of PEAP or EAP-TTLS authentication. The `radiusd.conf` file references the `sql.conf` file.
- **`users`** The traditional RADIUS configuration file for users. This file format is similar to the format defined in 1993. The `files` file references the `users` file.

4.0.1 Subdirectories of raddb

The `raddb` directory includes a number of sub-directories to help organize the configuration files:

- **`certs/`** This sub-directory stores EAP certificates.
- **`mods-available/`** A directory to store module configurations files that network administrators can enable on an optional basis. Each module is configured in a different file in the directory. The files in this directory are loaded from `radiusd.conf`.
- **`mods-enabled/`** A directory to store module configuration files. These are usually soft links to files in the `mods-available/` directory.
- **`sites-available/`** A directory to store virtual servers that administrators can enable. Each virtual server encapsulates one logical set of functionality.
- **`sites-enabled/`** A directory to store virtual servers. These are usually soft links to files in the `sites-available/` directory.
- **`sql/`** A directory to store more SQL configuration files for various types of databases, such as MySQL, PostgreSQL, or Oracle. There are sub-directories for each SQL database type.

This list identifies a number of other commonly used configuration files:

- **`acct_users`** A users file for Accounting-Request packets.
- **`hints`** The traditional RADIUS configuration file for giving PPP or SLIP hints as part of a User-Name.
- **`huntgroups`** The traditional RADIUS configuration file for groups of clients.

4.0.2 Overall Layout

The FreeRADIUS file layout described in this section is based on convention and is neither required nor enforced by the server. This file layout makes it easier to find a specific piece of the configuration, as each piece is broken into logically related components that are all grouped together in a sub-directory.

Nearly all of the files listed here are logically part of the main `radiusd.conf` file. Earlier versions of FreeRADIUS had all of the configuration files in one large `radiusd.conf` file, totalling nearly 2000 lines of text. This aggregation of files meant that understanding the configuration or finding specific portions of it was difficult for network administrators. The sub-directories listed in section 4.0.1 were introduced in version 2.0.0of FreeRADIUS. The `radiusd.conf` file is thus much smaller in version 2.0.0 and in all subsequent versions of FreeRADIUS.

4.1 The radiusd.conf File

The `radiusd.conf` file contains the server configuration. When the server starts, it reads this file and caches it. The data is parsed to set values for variables or to determine other configuration, such as modules.

Once the configuration is loaded, the server receives and processes packets. When the server is running in debugging mode (`radiusd -X`), the configuration that is being used is printed to the current terminal window.This information includes details about files being read, modules being loaded, and the names and values of any settings used.

The variables and subsections in the default `radiusd.conf` file and how they define specific functionality are discussed in more detail in the following section. To conserve space, this document describes only the variables that are commonly modified; the `radiusd.conf` file contains many more variables, which are documented in the comments section next to the variable definitions.

4.1.1 Variables

The default `radiusd.conf` file contains a large number of variables (see the comments section in the default configuration file for information about variables). In general, system administrators do not need to edit or change any of the variables not discussed in this section.

- **user** Often set to `radiusd`. The configuration files should be readable, but not writable by this user.
- **group** Often set to `radiusd`. The configuration files should be readable, but not writable by this group.
- **max_requests** When the server receives a request, it places the request into an internal queue for processing. At some later time, the server processes the request and sends a response. The requests that do not have a response are tracked through an internal counter. The `max_requests` variable prevents this counter from getting too large. This feature prevents the server from being overloaded if it receives a sudden burst of traffic.

  When the `max_requests` variable is set too low, the server discards new requests as soon as it becomes busy. The server also prints a descriptive error message to the console saying why the request is being discarded, and suggests that the network administrator increase the value of `max_requests`.

  When the `max_requests` variable is set too high, the server uses more memory for no real benefit. In general it is better for the network administrator to set this value high rather than too low.

  Setting the `max_requests` variable to zero (0) means there is no limit to the number of responses tracked by the counter. Setting `max_requests` to zero is not recommended, because of the potential server overload.

  Since each client sends the server streams of packets, network administrators should set the value of this variable according to the total number of clients. Thus, it is reasonable to multiply the total number of clients by 256, and then set the `max_requests` variable to that value. A future version of the server may use an automatic setting for this variable.

4.1.2 Log Subsection

The log subsection defines how the system handles the main server log messages (the server core produces these messages).

- **destination** This variable controls where the server sends the log messages. Set the destination to one of the following:
  - **files** Log to a flat-text file, where the filename is given by the `file` directive below.
  - **syslog** Log to syslog using the facility given by the `syslog_facility` directive below.
  - **stdout** Standard output. The use of `stdout` is recommended only when using Daemontools.
  - **stderr** Standard error.
- **file** The name of the file where the server writes all of the log messages. The default is `${logdir}/radius.log`. This file is re-opened on HUP so that it can be rotated.
- **syslog_facility** The values permitted here are dependent on the Syslog facilities defined by the local operating system. See `man syslog` for details. The default is `daemon`, which is common across all operating systems.
- **auth** Logs a summary line for each successful or failed authentication. This configuration may, in future, be removed because it duplicates the `linelog` functionality.
  The related variables `auth_badpass` and `auth_goodpass` control whether or not the passwords are also logged. For security purposes, the default is no. System administrators can set `auth_badpass=yes`, so that the log contains the passwords used in the failed authentication attempts. The administrator can then look at the log file to see exactly what the user entered.
- **msg_goodpass** This customizable message is appended to the summary log message that the server sends when `auth=yes`. This message dynamically expands and can contain information from any attribute.
- **msg_badpass** This customizable message is appended to the summary log message that the server sends when `auth=yes`. This message dynamically expands and can contain information from any attribute. It is most useful for logging specific reasons why the user was rejected, such as "Users in group X are not allowed to use NAS Y".

4.1.3 Security Subsection

The security subsection contains variables that affect the security of the server. While most other variables contain information such as file permissions, security subsection variables instead control how the server responds to perceived attacks.

- **max_attributes** Each request normally contains a small number of attributes. However, the packet format allows for malicious clients to send over 1000 attributes in a packet. Since each attribute is parsed into an internal data structure, these malicious packets can result in the server allocating up to a megabyte of memory for each request. If the malicious client sends a large number of requests, the server can quickly run out of memory.

  The "max_attributes" value thus limits the number of attributes per request. Requests that contain more than the "max_attributes" value are discarded, and cause the server to log a warning message. The default value for this variable is set to 200, which is sufficient for all normal operations. If the `max_attributes` warning message appears, contact the administrator to determine why so many attributes are being sent.
- **reject_delay** While it is generally useful to respond as quickly as possible to authentication requests, this fast response time allows attackers attempting to guess a password to perform tens of thousands of attempts per second.

  The `reject_delay` parameter controls how long the server waits before sending any Access-Reject. This delay prevents attackers from using a dictionary attack to guess passwords. This value is generally set to one (1). Leaving this value at 1 is recommended. Be aware that if you change this value to zero (0), the resulting fast response time would enable malicious attackers to guess passwords.
- **status_server** Administrators find it useful to know if the server is alive and responding to requests. The `status_server` configuration allows the server to respond to Status-Server requests. The response is either an empty Access-Accept or an Accounting-Request. RFC 5997 (www.ietf.org/rfc/rfc5997.txt) recommends using it in all clients and servers.

  Proxies commonly use this variable when they are trying to determine whether or not a home server is responding to requests. If the proxy suspects that the home server is unresponsive, it starts pinging the server with Status-Server requests. When the home server responds to these requests, the proxy marks it responsive.

  We recommend always setting this variable to `yes`. Setting it to `no` means that any client will have a more difficult time determining whether or not a server is responding, and thus authentication requests or accounting data may be lost.

4.1.4 Thread Subsection

The `thread` subsection defines parameters associated with child threads. The servers' main loop uses one thread to receive requests and place them into a queue. A pool of child threads then obtains requests from the queue and processes them.

The server can thus perform a number of tasks simultaneously, without blocking any tasks. For example, when a thread needs to read a field from a database, it may block and wait until the database responds. In the meantime, other threads can run and process other requests in parallel. This behavior increases the performance and robustness of the server.

However, as with all optimizations, there are also trade-offs. The server uses a number of variables, modeled after similar directives utilized by the Apache web server, to control and manage the trade-offs. The list below details these variables:

- **start_servers** This variable defines the number of threads started when the server first begins running.
- **max_servers** The `max_servers` variable controls the number of threads allowed to run at the same time. This variable is an upper limit.

  We recommend setting this variable high, rather than low. If the setting is too low, the requests remain queued until a thread becomes available, even if the CPU is not busy. This delay can result in a situation where requests are ignored until they time out.

  A message in debugging mode may appear with the following text:

  ```
  Thread spawn failed. Maximum number of threads (32) already running.
  ```

  This message may indicate that the value of the max_servers variable needs to be increased. On the other hand, the thread spawn failed message may also be caused by a slow server. This message may appear when all of the threads are waiting for the database to return. When a slow server is the cause of the problem, then increasing `max_requests` will not solve the issue; rather, the increase will cause more threads to wait in queue for the database.
- **{min,max}_spare_servers** These variables control the number of spare, or idle, threads. Since it takes time to start a new thread, it is a good idea to have a few idle threads waiting for requests. When a sudden spike of traffic occurs, the number of idle threads decreases. The variable then spawns more threads until `min_spare_servers` are idle. When the spike of traffic dies down, the number of idle threads increase. If it is more than `max_spare_servers`, some of the idle threads are shut down, until there are `max_spare_servers` idle.
- **max_requests_per_server** This variable is deprecated. It is recommended that this variable not be used. Leave this value at zero (0).

4.1.5 Modules Subsection

For versions prior to 2.0, the modules subsection contains the configuration for all the modules loaded by the server. In 2.0 and subsequent versions, the configuration for the modules is located in separate files in the `raddb/modules` directory. Use of the 2.0 configuration is recommended, as is keeping this section as small as possible.

A module definition is just another configuration section that contains variables:

```
module {
      variable = value
}
```

The module is made up of the module name, such as `pap`, `chap`, or `detail`. Each module configuration contains zero or more `variable` definitions. The definitions are specific to that module and are not shared across different modules.

Modules may also have subsections:

```
module {
      variable = value
      foo {
            bar = hello
      }
}
```

The subsections are specific to a particular module and are not shared across modules.

Modules can have specific *instances*. A module *instance* is a version of the module that shares the same functionality but uses a different configuration. For example, the `sql` module connects to SQL databases. If the server needs to connect to two different SQL databases, there must be two `sql` module instances. For example:

```
sql {
   driver = "rlm_sql_mysql"
   dialect = "mysql"
   server = "localhost"
   ...
```

To distinguish between the two databases referenced above, the first module can be referred to as `mysql` and the second module as `postgresql` (instead of just `sql`) by the system administrator. Any reference to an `sql` module causes the server to look for a configuration without an instance name. For example:

```
sql {
      database = ...
      server = ...
      ...
}
```

In this example, `sql`, `mysql`, and `postgresql` refer to three different instances of the `sql` module.

4.1.6 Instantiate Subsection

The `instantiate` subsection controls the order in which the modules are loaded into the server. Usually there are no inter-module dependencies, in which case the server can load the modules in any order. However, if there are dependencies, order can matter.

When module instantiation order matters, the names of modules to be loaded must be listed in the correct order in the this subsection. In the example below, the server would load the three modules named in this list sequentially (in the order in which they appear in the list):

```
sql
mysql
postgresql
```

The `instantiate` subsection also directs the server to load modules that are not directly referenced in any other section.

4.2 File Format

The file format is built around three basic elements: variable assignment, variable and module references, and sections.

The format of the configuration files is line-based text. Each configuration setting must be on a separate line. This format should be familiar to anyone who has seen other Unix configuration files.

Comments are allowed and are easily identified with the use of the # (hash) character: any text following a # character is considered to be a comment and is thus ignored by the parser. Spaces, tabs, and blank lines do not have any meaning and are also ignored by the parser.

Unintentional line breaks can be canceled by placing a back-slash (\) as the last character on the broken line. In the following example, the backslash after the equals sign forces both the first and second lines onto one continuous line:

```
foo = \
bar
```

The above example is then interpreted as:

```
foo = bar
```

4.2.1 Variable assignment

FreeRADIUS usually has predefined variable names with predefined meanings. For example, the variable `raddbdir` defines the directory where the server stores the configuration files. Variables are assigned values. The values can be integers, strings, file names, etc.

Multiple formats can be used to assign strings to variables. As with most systems, single and double-quoted strings can be used. Spaces and other characters can also be used in strings. For example:

```
secret = testing123
secret = "testing 1 2 3"
secret = 'testing 1 2 3'
```

The first example above sets the `secret` variable to `testing123`. The second and third example set it to `testing 1 2 3`, with embedded spaces.

The difference between the second and third example is that the double quotes mean that the string can be dynamically expanded, while the single quotes around the string means the value is taken "as-is" and not expanded.

Dynamic expansion is most useful when utilizing inter-variable references (see section 4.2.2, Variable References). For example:

```
foo = bar # sets variable foo' to value 'bar'
other = "my ${foo}" # sets variable 'other' to value 'a bar'
third = 'my ${foo}' # sets variable 'third' to value 'my ${foo}'
```

4.2.2 Variable References

Variable references are used when multiple variables contain the same value. In this example, the default configuration (first line) defines a `logdir` variable. The location of various log files in that directory are then cited in relation to the `logdir` variable reference:

```
logdir = /var/log/radius
...
file = ${logdir}/radius.log
...
detailfile = ${logdir}/detail
```

4.2.3 Module References

Module references either cause the server to load a shared module or call the module when the server receives a packet. These references are used by the server in only select locations (e.g., Authorize or Authenticate subsections) and cause errors when used elsewhere.

4.2.4 Sections

Sections define logical groups of configuration variables. Network administrators give each section a name and enclose the group using the { } characters:

```
mygroup {
      foo = bar
      other = "my ${foo}"
}
```

The previous example defines a simple section named 'mygroup' that contains two variables, 'foo' and 'other'. Like variable names, section names are also assigned specific meanings within the FreeRADIUS system. For example, the section name `client` defines the set of variables that are associated with a particular RADIUS client.

To define multiple clients, a `client` section is created for each client. A second qualifier, also known as an "instance name", is added to the section to distinguish the different sections of the same type. The following example shows two clients called foo and bar:

```
client foo {
      ipaddr = 192.0.2.100
      secret = testing123
}
client bar {
      ipaddr = 192.0.2.200
      secret = very_secret
}
```

There are a number of section type names, including `client`, `home_server`, `home_server_pool`, and `realm`, that the server will recognize. Specific to each section type are a list of variables that can be used inside that section. Examples showcasing these concepts can be found in the default configuration; the section name, function, and variable assignment are documented in the comments around each line.

4.2.5 Including a File by Reference

A configuration file can be split into multiple pieces and then be re-assembled at load time by using an `$INCLUDE` directive. This directive causes the configuration to include the particular file at the current

location, as if the referenced file was inserted in the file in place of `$INCLUDE`. For example, if we have a file called `other.conf` that contains the text:

```
other = "my ${foo}"
```

then the following two configurations are identical:

```
1.      foo = bar
        other = "my ${foo}"

2.      foo = bar
        $INCLUDE other.conf
```

Example 2 above uses an `$INCLUDE` to pull in the `other.conf` file, which results in a configuration that is identical to example 1 above.

Network administrators use the `$INCLUDE` directive to organize configuration files. Included files that begin with a `/` are absolute paths. Otherwise, the path is relative to the current file. Table 4.2.4 illustrates the difference in code.

Path Differences	
Absolute	**Relative**
`$INCLUDE /etc/raddb/foo.conf`	`$INCLUDE example/foo.conf`

Table 4.2.4 Path Differences

4.3 The Packet Processing Sections

The `Processing sections` form the engine that drives the server policies. When the server receives a request or sends a response, one or more processing sections are executed. The behavior of the server and the information that goes into any response is determined solely by how the sections have been processed.

The processing of a request depends on the following four pieces of information:

- The data associated with the request,
- the contents of the individual processing section,
- an action table associated with each processing section, and
- the processing of the algorithm itself.

The contents of the request processing section are Unlang statements, which have been described elsewhere (see "`man unlang`").

4.3.1 Terminology and definitions

This section describes the processing algorithm used by the server. For a basic installation, there is no need to understand this section in detail.

Requests

A number of attribute lists are associated with each request that is received by the server. Each list is named. The meaning of these list names is given in the table below.

Attributes are referred to by the list name, a colon, and then the attribute name (for example, request:User-Name, or reply:Class). Each list serves a particular purpose. Individual modules may edit one or more lists; for example, the mschap module will add MS-CHAP attributes to the reply. Alternatively, the administrator can use "unlang" statements to edit the lists directly.

request	Contains attributes that were received in the request
reply	Contains the attributes that will be sent in any reply
control	Contains "internal" attributes that are used by the server as a temporary storage area. None of these attributes are sent in a request or reply
proxy-request	Contains the attributes that are sent in any proxied request. Its contents are taken from the "request" list before any proxying is performed
proxy-reply	Contains the attributes that have been received from a home server and is used to re-initialize the "reply" list
coa	Valid only for Access-Request and Accounting-Request packets. Used to create a CoA-Request packet that is sent to the NAS. The word "disconnect" can be used instead, causing the server to send a Disconnect-Request packet to the NAS
coa-reply	Valid only for Access-Request and Accounting-Request packets. Used when receiving a CoA-ACK or CoA-NAK packet after sending a CoA-Request packet to the NAS. The word "disconnect-reply" can be used instead, referring to the attributes in any reply to a Disconnect-Request packet.

Table 4.3.1.1 Attribute lists associated with requests.

Processing Sections

Each request is passed through one or more processing sections. The sections and their meaning are given in the table below. Additional description is given in Section 4.3.2, below.

`authorize`	Access-Request packets are processed through the authorize section in order to obtain "known good" passwords prior to authentication.
`authenticate`	Access-Request packets are processed through the authenticate section, where a module implements a particular authentication method.
`post-auth`	Access-Request packets are processed through the post-auth section before a reply is sent to the NAS.
`preacct`	Accounting-Request packets are processed through the preacct section to normalize them prior to accounting being performed.
`accounting`	Accounting-Request packets are processed through the accounting section to perform accounting logging to files, SQL, and other accounting functions.
`pre-proxy`	All packets are processed through the pre-proxy section prior to being sent to a home server.
`post-proxy`	All packets are processed through the post-proxy section after a reply is received from a home server.
`recv-coa`	CoA-Request and Disconnect-Request packets are processed through the recv-coa section when they are received from an NAS.
`send-coa`	CoA-Request and Disconnect-Request packets are processed through the recv-coa section before a reply is sent to a NAS

Table 4.3.1.2. Processing sections and their meaning.

Each processing section is a list of modules to execute or Unlang statements to process. The list is processed in order from top to bottom.

In some cases, it is useful to skip parts of the list or to return early from processing a list. For example, if a “reject” is sent because of a policy rule, there is usually no reason to continue to process that list.

When a module is executed, a return code such as "reject", "noop", or "ok" is the result. These returns codes control processing of the list. The table below shows the names of the return codes and their meanings.

`notfound`	information was not found
`noop`	the module did nothing
`ok`	the module succeeded
`updated`	the module updated the request

`fail`	the module failed
`reject`	the module rejected the request
`userlock`	the user was locked out
`invalid`	the configuration was invalid
`handled`	the module has handled the request itself

Table 4.3.1.3. Return codes and their meaning.

Action Table

Each processing section has a list of default responses to the various return codes - for example, “noop” is usually ignored, “reject” causes the server to stop processing the list, etc. Each return code also has a priority: a "reject" has a higher priority than an earlier "noop".

The list of return codes and default actions is given in the table below.

Code	Action
`default`	`noop`
`reject`	`return`
`fail`	`return`
`ok`	`continue (priority 3)`
`handled`	`return`
`invalid`	`return`
`userlock`	`return`
`notfound`	`continue (priority 1)`
`noop`	`continue (priority 2)`
`updated`	`continue (priority 4)`

Table 4.3.1.4 Action table for most processing sections.

Algorithm

All of the pieces that are required to understand how the server processes a request have now been presented.

The server starts off each request with a default return code and priority. It processes the request through a processing section.

Each statement or module is evaluated, and the statement return code is used to update the final return code associated with the request. When the list is finished, the server either continues to the next section or sends a reply to the request.

The algorithm used by the server is given in the following pseudo-code:

```
 (code, 0) = action_table[default]
 foreach (statement in section) {
  code' = evaluate(statement)
  (action, priority') = action_table[code']
  if (action == return) {
    code = code'
    break;
  }
  if (priority' >= priority) {
   (code, priority) = (code', priority')
  }
 }
 return (code, priority)
```

The "evaluate" function in the above algorithm should be seen as either executing a module (e.g. mschap) or as interpreting an "unlang" statement. Where an unlang statement involves a sub-section, the algorithm is run recursively on that sub-section.

The following text is advanced level information that documents how the server processes the packets. The following section is not needed for basic installation.

4.3.2 Server Processes

The server processes each request through one or more packet processing sections. Running FreeRADIUS can involve any of the following processing sections:

- authorize
- session
- authenticate
- post-auth
- preacct
- accounting
- pre-proxy
- post-proxy
- send-coa
- recv-coa

Figure 4.3 illustrates the various packet processing sections that are utilized by different operations.

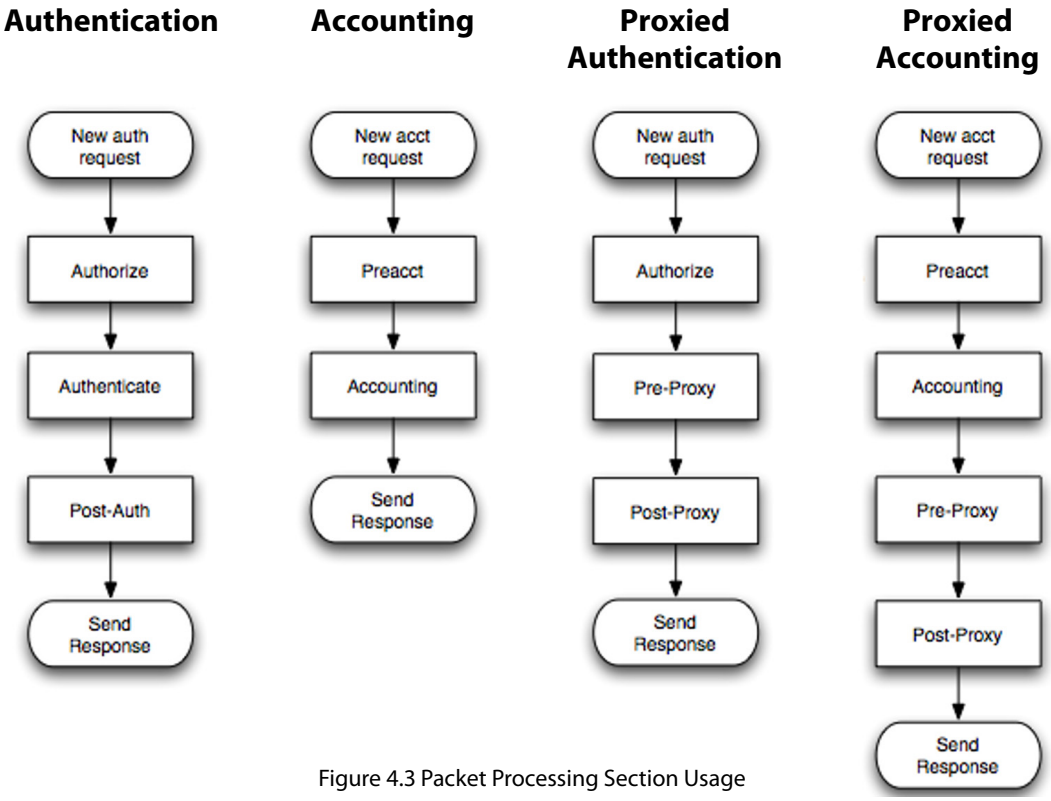

Figure 4.3 Packet Processing Section Usage

4.3.3 Authentication

This section describes what happens when the server receives an access request packet and authenticates a user.

The Authorize Section

The name of this section is `authorize` for historical reasons, as earlier versions of the server did not have a post-auth section. A more accurate description of this section would be `pre-authentication`.

The authorize section processes `Access-Request` packets by normalizing the request, determining which authentication method to use, and either setting the "known good" password (the valid password found in the database) for the user or informing the server that the request should be proxied.

Once the authorize section has finished processing the packet, the return code for the section is examined by the server. If the return code is `noop`, `notfound`, `ok`, or `updated`, then request processing continues. If the return code is `handled`, then it is presumed that one of the modules set the contents of the reply, and the server sends the reply message. Otherwise, the server treats the authentication as being rejected and runs the post-auth section.

If the authentication has not been rejected, then the server continues processing the request by searching for the `Auth-Type` attribute in the `control` list. Once the `Auth-Type` attribute is found, then the named sub-section of `authenticate` is executed, as described below. This functionality is historical and dates back to versions prior to 2.0.0. For versions 2.0.0 and later, we recommend using Unlang policies, which are more flexible and simpler to understand.

NOTE: The authorize section should not be used to set attributes in a reply. Although this practice is widespread and is in the default configuration for historical reasons, it is not good policy design.The authorize section should be used to define a policy and the post-auth section should be used to set the reply attributes.

The Session Section

The session section is used to perform database lookups when enforcing `Simultaneous-Use` or double login detection. It is used only for `Access-Request` packets.

Code	Action
`default`	`fail`
`reject`	`return`
`fail`	`continue (priority 1)`
`ok`	`return`
`handled`	`return`
`invalid`	`return`
`userlock`	`return`
`notfound`	`return`
`noop`	`return`
`updated`	`return`

Table 4.3.3.1 Action table for the session section.

The Authenticate Section

The `authenticate` section is only used when the server is authenticating requests locally and is bypassed completely when proxying. This section is different from each of the other sections: it is composed of a series of subsections, only one of which is executed. The relevant subsection is chosen based on the contents of the `Auth-Type` attribute found in the `control` list.

The `Auth-Type` attribute can also refer to a module (e.g. `eap`) instead of a subsection, in which case that module, and only that module, is processed.

The `Auth-Type` subsection may contain a series of Unlang statements, which is useful when modifying the results of authentication.

Once the `authenticate` section has finished, then the post-auth section is executed.

Code	Action
`default`	`reject`
`reject`	`return`
`fail`	`continue (priority 1)`
`ok`	`return`
`handled`	`return`
`invalid`	`continue (priority 1)`
`userlock`	`return`
`notfound`	`return`
`noop`	`continue (priority 1)`
`updated`	`continue (priority 1)`

Table 4.3.3.2 Action table for the `Authenticate` section.

The Post-auth Section

The post-auth section contains policies that are applied after the authentication process either succeeds or fails (is rejected).

When authentication succeeds, the contents of the post-auth section are processed, along with any other relevant section.

When authentication fails, the server looks for a subsection called `Post-Auth-Type Reject`, which, if found, executes only the statements that are found within that section. If no such subsection exists, then no post-auth processing takes place.

To update any reply attributes, it is recommended to only use the post-auth section, although this may be difficult. Many modules provide reply attributes only in their `authorize` method. That method can be called from the post-auth section by using the `Module-name.authorize` syntax (e.g., `sql.authorize`).

4.3.4 Accounting

This section describes what happens when the server receives an accounting request.

The Preacct Section

The preacct section is used to process `Accounting-Request` packets by normalizing the request and determining whether or not the request should be proxied.

If the section returns one of `noop`, `ok`, or `updated`, then the accounting section is processed. Otherwise, the server stops processing the request and does not respond with an `Accounting-Response` packet.

Code	Action
default	noop
reject	return
fail	return
ok	continue (priority 2)
handled	return
invalid	return
userlock	return
notfound	return
noop	continue (priority 1)
updated	continue (priority 2)

Table 4.3.4.1 Action table for the preacct section.

The Accounting Section

The accounting section is used to process `Accounting-Request` packets.

If the `Acct-Type` attribute is set in the `control` list, then only that named subsection is processed. Otherwise, the entire accounting section is processed, and `acct-type` sections are ignored.

The server checks if the request should be proxied once the accounting section is processed. This sequence allows the server to log accounting packets locally and to proxy them at the same time..

Code	Action
default	noop
reject	return
fail	return
ok	continue (priority 2)
handled	return
invalid	return
userlock	return
notfound	return
noop	continue (priority 1)
updated	continue (priority 3)

Table 4.3.4.2 Action table for the accounting section.

4.3.5 Proxied Authentication

This section describes what happens when the server receives an access request and then proxies it.

The Authorize Section

During proxied authentication, the authorize section is run the same as per the normal authentication process. Please see section 4.3.3 for further information on this section.

The Pre-proxy Section

The pre-proxy section is used to process a request before it is proxied. This section is used for all requests, including `Access-Request`, `Accounting-Request`, `CoA-Request`, and `Disconnect-Request`.

If different policies are needed for different types of packets, the `home_server_pool` section can be used to define a `virtual_server` entry. That virtual server can then contain pre-proxy and post-proxy sections, which are executed only when sending requests to the home servers in that pool.

Since home servers are typed and are tied to the type of request being sent, this configuration allows different policies for each type of request.

The Post-proxy Section

The post-proxy section is used to process a reply from a home server. As with the pre-proxy section, this section is used for all requests, including `Access-Request`, `Accounting-Request`, `CoA-Request`, and `Disconnect-Request`.

The post-proxy section is most commonly used to filter replies from home servers in order to remove inappropriate authorizations such as VLAN settings or IP address assignments.

As with the pre-proxy section, the `home_server_pool` section can be used to define a `virtual_server` entry if different policies are needed for different types of packets.That virtual server can then contain pre-proxy and post-proxy sections, which are executed only when sending requests to the home servers in that pool.

The `home server` may not respond to a proxied request within the `response_window` time (see the `proxy.com` file). When the `response_window` time expires, the server that proxied the request then executes the `Post-Proxy-Type Fail` subsection of the post-proxy section. This behavior allows the administrator to define a policy for an unresponsive home server.

Once the post-proxy section has finished executing, any existing attributes in the `reply` list are discarded, and the post-proxy attributes are copied to the `reply` list. This behavior allows a home server to define the default reply sent back to the NAS.

4.3.6 Proxied Accounting

The Preacct Section

During proxied accounting, the preacct section is run the same as during the normal accounting process. Please see section 4.3.4 for further information on this section.

The Accounting Section

During proxied accounting, the accounting section is run the same as during the normal accounting session. Please see section 4.3.4 for further information on this section.

The Pre-proxy Section

During proxied accounting, the pre-proxy section is run the same as during proxied authentication. Please see section 4.3.5 for further information on this section.

The Post-proxy Section

During proxied accounting, the post-proxy section is run the same as during proxied authentication. Please see section 4.3.5 for further information on this section.

4.3.7 Change of Authorization (CoA)

The Send-coa Section

The send-coa section is responsible for sending Change of Authorization messages. These messages are sent to the NAS, which then disconnects the user from the network. Network administrators can also use Change of Authorization messages to add additional filter rules to a user's session; an example would be setting a bandwidth limit on a user's session.

The Recv-coa Section

The recv-coa section is used to process `CoA-Request` or `Disconnect-Request` packets from an NAS. This section can cause the request to be proxied, in which case the pre-proxy and post-proxy sections are executed, as described above.

The recv-coa section receives and processes Change of Authorization messages. Modules in this section can display messages if, for example, a user is disconnected.

4.3.8 Minimal radiusd.conf

The default configuration files contain thousands of lines of configuration definitions and comments. While all of this data may seem imposing at first, the server configuration can be very simple. The default configuration is complex because it does simple accounting and supports many authentication protocols.

This section provides an example of a small simplex configuration file to show just how simple the server configuration can be. The only requirements are for the server to:

- Listen on an IP address and port
- Accept packets from a known client
- Not use any plug-in modules
- Set the authentication to Accept

An example simple configuration is shown below. Place the following text in a file named `small.conf`:

```
listen {
      type = auth
      ipaddr = *
      port = 0
}
client localhost { # allow packets from 127.0.0.1
      ipaddr = 127.0.0.1
      secret = testing123
}
modules { # We don't use any modules
}
authorize { # return Access-Accept for PAP and CHAP
      update control {
            Auth-Type := Accept
      }
}
```

To start the server in debugging mode, use the `small.conf` file instead of the default `radiusd.conf` file. You can do this by using the following command:

```
$ radiusd -X -n small
```

This configuration takes any localhost `Access-Request` that contains a PAP or CHAP authentication method and returns an `Access-Accept`.

4.4 The Users File

The `files` module authorizes each user request via the `users` file. The `users` file is located in the RADIUS database directory.

The users file treats every line starting with a hash sign (' # ') as a comment to be ignored.

Each entry consists of alternating check item and reply item lines.

Each item in the check or reply item list is an attribute in the form `name = value`. Multiple items can be placed on each line. If there is more than one item on each line, the items must be separated with commas.

The file begins with a check item line: a username followed by a (possibly empty) list of check items. The check item line must be confined to one line.

Then the reply item line begins with a tab and a (possibly empty) list of reply items. The reply items can be specified over multiple lines; however, each line - except the last line - must end with a comma. The last line of the reply items must not end with a comma.

The check items are a list of attributes that the server uses to match the incoming access requests. If the username and all of the check items match the incoming access request, the server adds those attributes to the reply. This process is repeated for all of the entries in the users file.

If the incoming request does not match all of the check items in any one entry, the `files` module returns a `notfound` reply, stating that it could not find a record of the user.

An example of a `users` file entry is below. It states that a user "bob" exists, with a password "hello". When the user "bob" logs in, a reply message saying "Hello, bob" will be received by the user.

```
bob     Cleartext-Password := "hello"
        Reply-Message := "Hello, %{User-Name}"
```

The special username `DEFAULT` *matches all usernames*.

Entries are processed sequentially, starting from the top of the `users` file. If an entry contains the special item `Fall-Through = No` as a reply attribute, then the processing of the file stops at this point, and no more entries are matched. If a reply item list does not contain a `Fall-Through` attribute, it is treated as though it included a default `Fall-Through = No` attribute. If an entry contains the special item `Fall-Through = Yes` as a reply attribute, then processing proceeds to the next entry.

`Fall-Through` should be used carefully and the server should be tested in debugging mode with a number of test requests to verify that the configured entries behave as expected.

4.4.1 Operators

Operators other than = can be used for the attributes in either the check item, or reply item list. The following is a list of operators and their definitions.

- **`Attribute = Value`**
  Although it is not allowed as a check item for RADIUS protocol attributes, this operator can be used for server configuration attributes (for example, Auth-Type). It sets the value of an attribute only if there is no other item of the same attribute.
  As a reply item, it is interpreted to mean "add the item to the reply list, but only if there is no other item of the same attribute."
- **`Attribute := Value`**
  Always matches as a check item. In the configuration items, it replaces any attribute of the same name. If no attribute of that name appears in the request, then the attribute is added.
  As a reply item, it has an identical meaning, but for the reply items.
- **`Attribute == Value`**
  As a check item, it matches only if the named attribute is present in the request AND has the given value. Not allowed as a reply item.
- **`Attribute += Value`**
  Always matches as a check item and adds the current attribute with value to the list of configuration items.
  As a reply item, the attribute is added to the reply items.
- **`Attribute != Value`**
  As a check item, it matches only if the given attribute is in the request AND does not have the given value. Not allowed as a reply item.
- **`Attribute > Value`**
  As a check item, it matches if the request contains an attribute with a value greater than the one given. Not allowed as a reply item.
- **`Attribute >= Value`**
  As a check item, it matches if the request contains an attribute with a value greater than or equal to the one given. Not allowed as a reply item.
- **`Attribute < Value`**
  As a check item, it matches if the request contains an attribute with a value less than the one given. Not allowed as a reply item.
- **`Attribute <= Value`**
  As a check item, it matches if the request contains an attribute with a value less than or equal to the one given. Not allowed as a reply item.
- **`Attribute =~ Expression`**
  As a check item, it matches if the request contains an attribute that matches the given regular expression. This operator may only be applied to string attributes. Not allowed as a reply item.
- **`Attribute !~ Expression`**
  As a check item, it matches if the request contains an attribute that does not match the given regular expression. This operator may only be applied to string attributes. Not allowed as a reply item.
- **`Attribute =* Value`**
  As a check item, it matches if the request contains the named attribute, regardless of its value. Not allowed as a reply item.
- **`Attribute !* Value`**
  As a check item, it matches if the request does not contain the named attribute, regardless of its value. Not allowed as a reply item.

4.4.2 Examples

This section provides examples of FreeRADIUS code that can be included in the `users` file.

1. The following example tells the server that “bob” has a known good password "hello". There are no reply items, so the reply will be empty.

```
bob    Cleartext-Password := "hello"
```

2. This example tells the server to authenticate all users against the /edc/passwd file. Note that this example makes EAP authentication fail.

```
DEFAULT      Auth-Type = System
             Fall-Through = Yes
```

3. If the request packet contains the attributes Service-Type and Framed-Protocol with the given values, then those attributes are included in the reply. This example also shows how to specify multiple reply items.

```
DEFAULT Service-Type == Framed-User, Framed-Protocol == PPP
       Service-Type = Framed-User,
       Framed-Protocol = PPP,
       Fall-Through = Yes
```

See the `users` file supplied with the server for additional examples and comments.

4.4.3 Troubleshooting the Users File

The `users` file can be checked by running the server in debugging mode (`-X`) and using the `radclient` program to send it test packets to match specific entries. The server will print out matching entries for that request, so the expected outcome can be verified. Do this before doing anything else if problems with the file are suspected.

Be careful when writing entries for the `users` *file. It is easy to incorrectly configure the server to accept requests instead of rejecting requests. Order the entries and only use the* `Fall-Through` *item where required.*

Entries rejecting certain requests should go at the top of the file, and should not have a `Fall-Through` *item in their reply items. Entries for specific users who do not have a* `Fall-Through` *item should come next. Any default entries should come at the end of the file. When the server reaches a DEFAULT entry, it will stop processing unless it has* `Fall-Through` *set.*

Glossary of Terms

AAA - Authentication, Authorization, and Accounting - a security architecture for distributed systems that provides control over user access to services and resources and tracks user activities.

ACL - Access Control List - a list of permissions attached to an object. It specifies which users or system processes are granted access to objects and what operations are allowed on given objects.

ADSL - Asymmetric digital subscriber line.

AVP - Attribute value pair.

CHAP - Challenge-Handshake Authentication Protocol -a protocol that authenticates network users.

EAP - Extensible Authentication Protocol - an authentication framework. Variants like EAP-TTLS and EAP-TLS add the Transport Layer Security protocol.

FreeRADIUS - a modular, high performance, open source variant of RADIUS.

FTP - File Transfer Protocol - a standard network protocol used to transfer files from one host to another host over a Transmission Control Protocol-based network, such as the internet.

HTTP - Hypertext Transfer Protocol - an application protocol for distributed, collaborative, hypermedia information systems used on the World Wide Web.

IANA - the department of the Internet Corporation for Assigned Names and Numbers, which oversees global IP address allocation, autonomous system number allocation, root zone management in the Domain Name System, media types, and other Internet Protocol-related symbols and numbers.

IETF - Internet Engineering Task Force - an international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet.

ISP - Internet Service Provider (also called an Internet Access Provider) - a business or organization that offers users access to the Internet and related services.

L2TP - Layer 2 Tunneling Protocol - a tunneling protocol used to support VPNs or as part of the delivery of services by ISPs.

LAN - Local Area Network - a computer network that interconnects computers in a limited area such as an office building, school, or computer laboratory using network media.

LDAP - Lightweight Directory Access Protocol - an application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network.

MAC address - Media Access Control address - a unique identifier assigned to network interfaces for communications on the physical network segment.

NAC - Network Access Control - a set of protocols to define and implement a policy that describes how to secure access to network nodes by devices when they initially attempt to access the network.

NAS - Network Access Server - a single point of access to a remote resource, any device or application that performs username and password authentication.

PAP - Password Authentication Protocol - an authentication protocol that uses an unencrypted ASCII password.

PEAP - Protected Extensible Authentication Protocol - a protocol that encapsulates EAP in an encrypted and authenticated TLS tunnel.

POP - Point Of Presence - an access point to the Internet. It is a physical location that houses servers, routers, ATM switches and digital/analog call aggregators.

PPP - Point-to-Point Protocol - a data link protocol used for direct communication between two networking nodes.

QoS - Quality of Service - a network's ability to achieve maximum bandwidth and deal with network performance elements such as latency, error rate, and uptime.

RADIUS - Remote Authentication Dial In User Service - a network protocol for remote user authentication and accounting.

RFC - an Internet Engineering Task Force memorandum on Internet standards and protocols.

SLIP - Serial Line Internet Protocol - an encapsulation of the Internet Protocol designed to work over serial ports and modem connections.

SNMP - Simple Network Management Protocol - an Internet-standard protocol for managing devices on IP networks. Such devices include routers, switches, servers, workstations, and printers.

SSID - Service Set Identifier - a human-readable text string that identifies an extended service set.

SSL - Secure Sockets Layer - an asymmetric cryptographic protocol that provides communication security over the Internet. It is a predecessor of the TLS protocol.

SQL - Structured Query Language - a special-purpose programming language designed for managing data held in a relational database management system. Open source versions include MySQL and PostgreSQL.

TLS - Transport Layer Security- an asymmetric cryptographic protocol that provides communication security over the Internet. It is the successor to the SSL protocol.

TTLS - Tunneled Transport Layer Security - an extension to TLS that uses an established secure connection, or tunnel, to authenticate the client. The addition of the tunnel provides protection from eavesdropping and man-in-the-middle attack. It is used in EAP-TTLS.

UDP - User Datagram Protocol - a network communications method.

VLAN - Virtual Local Area Network - a logical group of workstations, servers, and network devices that function as if on the same LAN, regardless of geographical distribution.

VPN - Virtual Private Network - a private network that exists in a public network.

VSA - Vendor-Specific Attributes - attributes defined by remote-access server vendors, usually hardware vendors, to customize how RADIUS works on their servers.

Frequently Asked Questions

Which operating system should I use?

The FreeRADIUS protocol works on all Unix based systems. Selecting a familiar operating system is recommended. Windows does not currently support FreeRADIUS.

What are the CPU/RAM/disk space requirements for the server?

A FreeRADIUS server has minimal requirements. A basic FreeRADIUS installation uses 8 megabytes of RAM, under one hundred megabytes of disk space, and minimal CPU power. An Internet Service Provider with 10,000 or fewer users will not have any problems with any commodity system available at the time of this printing. If the ISP has more than 10,000 users, the overall system design becomes much more important than the specifications of an individual server.

The server isn't starting. What should I do?

Do the following:

- Use sudo to switch to the root context.
  If the server starts correctly, the problem is the server is running as an unprivileged user.
- Log in as that user, and run it in debugging mode. Any errors or warnings will be appear on the monitor.

The server is sending an Access-Reject message when the user should be accepted. Why?

Do the following to gather enough information to diagnose the problem:

- Run the server in debugging mode and observe what happens when the user logs in.
- If the control socket is configured, you can also run `raddebug -u USERNAME` to gather the same information from a running server.

The server is sending Access-Accept with some attributes, but the user is not getting the correct kind of service (or is getting rejected). Why?

Check the NAS logs to determine why the NAS is ignoring instructions from the server. The NAS documentation should identify the attributes it needs to provide certain services. These attributes can sometimes be in unusual formats, so be sure to consult the NAS documentation.

The accounting logs do not contain the information I want to see. How can I make the server log the correct information?

Configure the NAS to send the correct information. A FreeRADIUS server can only log the information it receives. If the information isn't in the accounting packets, then it wasn’t sent by the NAS.

How can I restrict a users network access after authentication?

Configure FreeRADIUS to send the appropriate filtering rules to the NAS. The NAS enforces these rules after authentication. Consult the NAS documentation for the names and format of the attributes. There are no accepted standards for these attributes, so be sure to consult the NAS documentation.

How can I configure a captive portal for users?

Consult the NAS documentation on how to configure a captive portal. Each NAS requires different action.

Another option is to use an Open Source captive portal solution. FreeRADIUS only provides a Yes or No response to authentication requests. FreeRADIUS does not act as a firewall to block network access and does not support HTTP.

How can I disconnect a user after they have logged in?

See the NAS documentation. If the specific NAS supports Disconnect-Request, you can use `radclient` to send that packet.

If the NAS doesn't support the Disconnect-Request, take the appropriate action specific to that particular NAS. Some NASs support SNMP writes, while other NASs require the administrator to log into the NAS console and forcibly reset the port.

The server logs have many numerous messages about request timeout or blocked child. How can I fix this?

These blockages generally occur because the server is using a database for authentication that is inactive or slow. To solve this problem, increase the database's response speed. As a rough guideline, if a query takes more than a second, something is catastrophically wrong with the database. Possible solutions include:

- Adding indexes
- Adding more RAM
- Making sure that other systems are not accessing the database when FreeRADIUS needs to use it

How many authentications per second can the server handle?

This is largely dependent on the CPU power of the host system, the type of database, and the authentication protocol. Typically, a new server with all users in memory and PAP authentication should be able to do many tens of thousands of authentications per second.

When using PEAP, the system can be CPU bound and may only do thousands of authentications per second. If the users are stored in a database, FreeRADIUS is limited to the speed of the database. For example, a database that performs a thousand queries per second limits FreeRADIUS to the same number.

How many accounting packets per second can the server handle?

The host system's CPU power and database determines how many accounting packets the server can handle per second. If the database is limited to a thousand writes per second, the FreeRADIUS accounting requests is also limited to a thousand writes per second.

How can I maximize my system's performance?

Make sure that your system is using a high performance database. This single action will have the greatest impact on your systems performance.

The server is giving me a WARNING / ERROR message. What should I do?

Read the warning or error message and follow the instructions. If the problem persists, ask for assistance on the general FreeRADIUS mailing list at **freeradius-users@lists.freeradius.org**.

The server says shared secret is incorrect, but I am sure it was entered correctly! How can I fix this?

To correct this issue, perform the following steps:

1. Re-enter the shared secret on the NAS and on the RADIUS server.
2. Run the server in debugging mode to verify that it is using the correct shared secret for that NAS.

In some cases, this message indicates that the NAS network hardware is broken. If this is the case, replace the NAS.

How do I get technical support help if I need it?

Network RADIUS SARL prides itself on providing the highest quality FreeRADIUS support available. Network RADIUS SARL started and maintains the FreeRADIUS protocol and writes the majority of new RADIUS specifications.

For more information on Network RADIUS SARL support packages, go to: **http://networkradius.com/support**. See the table below for a brief description of the options.]

Support Options			
Description	**Silver**	**Gold**	**Platinum**
Cost in Euros (per server per year)	2000.00	3000.00	5000.00
Hours of coverage	9 AM - 5 PM UTC Mon - Fri	24 / 7	24 / 7
Critical incident response time	-	4 hours	2 hours
Regular response time	8 business hours	8 business hours	2 business hours
Support method	email / web	email / web / phone (for critical incidents only)	email / web / phone

Support Options			
Preventative consulting	-	4 hours per month	8 hours per month
Security notifications	Yes	Yes	Yes
Bug notifications	Yes	Yes	Yes
Bug fixes	-	Yes	Yes
Technical account manager	Yes	Yes	Yes

How can I get training for my staff?

Network RADIUS SARL offers a range of on-site training courses to meet your needs. Courses include:

- **Introduction to RADIUS** An introductory course for all administrators who are unfamiliar with RADIUS. Basic RADIUS concepts that are key to understanding RADIUS-based systems are introduced.
- **Advanced RADIUS** A course for senior administrators who are already familiar with RADIUS. It introduces advanced RADIUS concepts that will help system administrators design, deploy, maintain and debug their systems.
- **Programming with RADIUS** A course for developers who wish to create their own customized solutions.

Contact **sales@networkradius.com** for more information on our training classes.

How can I get a consultant to help me?

Our goal at Network RADIUS SARL is to help you build a world-class system and to make sure it operates smoothly. Network RADIUS SARL are experts at designing a customized RADIUS solution that meet your needs. Your customized network will include the appropriate number of RADIUS and database servers for your particular business needs. We install the RADIUS server and database and create the necessary tables, schemas, queries, and replication.

We can configure multiple forms of authorization on the same system simultaneously. The system can include:

- 802.1x
- PEAP
- EAP-TTLS
- EAP-TLS
- Authentication against Active Directory
- MAC authentication
- MAC auth bypass (MAB)

Existing systems can be migrated to our product or we can configure our product to work with your databases. The final result is a system that is robust, high performance, and easy to maintain. Contact us at **sales@networkradius.com** for more information.

What are the estimated costs and times involved in an install and implementation?

Since every company's network requirements are unique, these can vary. The first step is to contact Network RADIUS SARL for a consultation. We will evaluate your organization's needs and design a system to match. This may include:

- Installing a new system
- Upgrading or customizing existing systems
- Migrating legacy RADIUS servers to FreeRADIUS

Contact us at **sales@networkradius.com** for more information or quotes.

Are there any recommended online resources such as user forums or groups to help with implementation questions?

There are several mailing lists associated with the FreeRADIUS server project. The lists can be found at **http://freeradius.org/list/**. The current lists are:

- **freeradius-users@lists.freeradius.org**
  This list is for all users of FreeRADIUS and deals with general questions related to FreeRADIUS
- **freeradius-devel@lists.freeradius.org**
  This list is for developers who are writing code for FreeRADIUS.The content is highly technical and is not suited to the average user.
- **freeradius-announce@lists.freeradius.org**
  This list is for all users of FreeRADIUS. Announcements about FreeRADIUS, including new versions and security issues, are made here.

What other documentation is available for the product?

While a great deal of information can be found online for FreeRADIUS, much of it is out-of-date or incorrect. Avoiding third party documentation is thus strongly recommended. Additional documentation created by Network RADIUS SARL includes:

- **The FreeRADIUS Reference Guide**
- **The FreeRADIUS Implementation Guide**

If I notice a problem or bug, how do I notify the development team?

A list of known issues can be found at **https://github.com/FreeRADIUS/freeradius-server/issues**. Read the list to see if your issue has been discovered and if a solution exists.

If no open issue matches what you're experiencing, create a new issue. Note that this will require creating a GitHub account. To create a new issue, do the following:

1. Click the New Issue button.
2. Follow the instructions on screen.

For any potential security issues, e-mail **security@freeradius.org**.