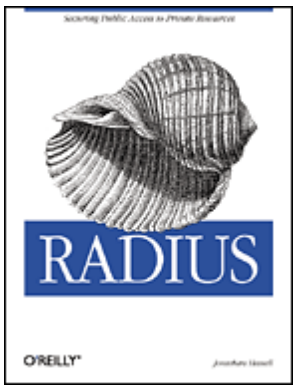




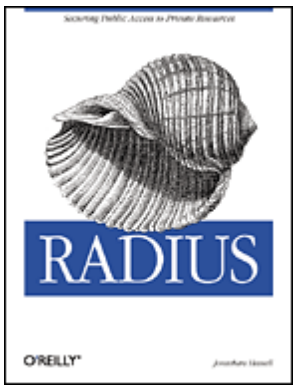

**RADIUS**

By Jonathan Hassell

START READING

Publisher: O'Reilly
Date Published: October 2002
ISBN: 0-596-00322-6
Pages: 206

RADIUS, or Remote Authentication Dial-In User Service, is a widely deployed protocol that enables companies to authenticate, authorize and account for remote users who want access to a system or service from a central network server. RADIUS provides a complete, detailed guide to the underpinnings of the RADIUS protocol. Author Jonathan Hassell brings practical suggestions and advice for implementing RADIUS and provides instructions for using an open-source variation called FreeRADIUS.






**RADIUS**

By Jonathan Hassell

START READING

Publisher: O'Reilly
Date Published: October 2002
ISBN: 0-596-00322-6
Pages: 206

# Copyright



Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

# Preface

*"Trust no one."*

Aside from being the motto and modus operandi for the successful TV series The X Files, it is also a beneficial mantra to practice in all facets of security and investigation. Even though chances are good that you won't encounter Mulder, Scully, and their gang, they still teach a concept that's become increasingly relevant as the world—and the world's computers—become connected.

Companies today are increasingly basing their business models around providing access to resources—web pages, Internet access, email accounts, or anything else—that need to be protected. How does a user indicate to a system, especially one that indeed trusts no one, that he's entitled to use that computer's services? How can the owner of a business keep non-paying users out of the way while providing convenient access to paying customers? The bottom line is this: with new security exploits being uncovered every day and the general environment of the Internet public degenerating from a trusted environment into one of hostility and attack, there has to be some way in which an Internet citizen can use resources to which he's entitled without letting everybody else in the gates.

This is the purpose of the RADIUS protocol—to differentiate, secure, and account for these users. And the purpose of this book is to provide the most complete reference to RADIUS possible.

# Audience

This book is designed to serve all manner of readers. There's enough introductory information to give a complete, generalized background for the administrator not familiar with the protocol. There's practical, day-to-day, hands-on information for those tasked with configuring and using RADIUS servers. There's design-level information for programmers who need to write custom applications to integrate RADIUS. In other words, there's something in this book for everyone.

# Organization

I have tried to structure this book as effectively as possible, mixing theory with practice where appropriate, so you, the reader, have a firm background with which to apply both the practical advice and procedures in this book and others you may develop on your own.

Chapter 1 takes a few steps backward and looks at the architectural model on which the RADIUS protocol is based, provides an introduction to RADIUS's characteristics and limitations, and offers a brief discussion of its history.

Chapter 2 details the individual characteristics of the RADIUS protocol, including an overview of its standard packet formats and the structure of the properties it passes to various servers, as well as a discussion of how vendors extend the functionality of the protocol through the use of their own defined attributes. There is also commentary on the various authentication protocols that can be used in conjunction with RADIUS, as well as a brief introduction to the hints file.

Chapter 3 is a reference section for all of the globally defined RADIUS attributes as specified in the appropriate RFC documents. An "at a glance" chart details each attribute's primary properties with a short discussion of its purpose. Any special behaviors that an administrator might encounter during its use are covered in this discussion.

Chapter 4 is presented as a combination of the stylistic elements of Chapter 2 and Chapter 3 and covers the properties, behaviors, and attributes of the accounting portion of the RADIUS protocol. It discusses standard accounting packets, proxy functionality, and the standard accounting attributes as specified by the RFCs.

Chapter 5 is the first hands-on chapter in the book. It discusses obtaining, installing, configuring, and using FreeRADIUS, an open source RADIUS server that was created in part by several developers of the Debian Linux distribution.

Chapter 6 continues the practical guidance and covers the more intimate and intricate configuration options that FreeRADIUS provides. In addition, extending FreeRADIUS's functionality is covered, by having it authenticate against a MySQL database, use the pluggable authentication module (PAM) in its transactions, and interact with Cisco networking gear. Simultaneous use, also known as multilinking in the ISP business, is also covered.

Chapter 7 discusses other programs to augment FreeRADIUS, including an Apache module that will allow the web server to authenticate against the RADIUS user database, a powerful email and directory server that will consolidate user information and reduce administrative headaches, and a utility for parsing and analyzing RADIUS log files.

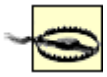
Chapter 8 is a commentary on some of the security problems the protocol has and how to work around them. Unfortunately, the protocol used to secure networks has some vulnerabilities of its own, and this chapter offers insight into what the vulnerabilities are, how they were introduced, and what an administrator can do to eliminate the potential threat they represent.

Chapter 9 includes information that's not present in the original RFC documents for the protocol. Among these new details are information on tunnel support, Apple networking support, interim accounting updates, using Extensible Authentication Protocol (EAP), and a listing—like that of Chapter 3—of the new attributes added by the RADIUS

# Conventions Used in This Book

- *Italic* is used for filenames, directories, URLs, emphasis, and the first use of technical terms.
- Constant width is used for IP addresses, configuration file operators, and packet names and attributes.
- **Constant width bold** is used for user input.

This icon designates a note, which is an important aside to nearby text.



This icon designates a warning relating to the nearby text.

# How to Contact Us

Please address comments and questions concerning this book to the publisher:
O'Reilly & Associates, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 (800) 998-9938 (in the United States or Canada) (707) 829-0515 (international or local) (707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:
http://www.oreilly.com/catalog/RADIUS

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

The author has created a comprehensive web site to support this book, located at http://www.theradiusbook.com. You can find an overview, the table of contents, a listing of errata, sample code, and many other resources at that site.

# Acknowledgments

A variety of people came together to make this book possible. My publisher, O'Reilly, has made the writing and production of this book proceed as smoothly as possible. I'd also like to extend heartfelt thanks to my editor, Jim Sumser. I've sat here for an hour attempting to come up with fifty words that could adequately describe what a genuinely fine man Mr. Sumser is, and I conclude it's not possible. Rarely in life are you provided an opportunity to work with such a gentlemen and professional. Jim defines those qualities, and I am better for it.

I'd also like to thank the fine folks at Equipment Data Associates (EDA) in Charlotte, North Carolina for offering me an opportunity to work and write. The flexibility I was granted in working with EDA was a godsend and delayed the onset of gray hair on me by at least two years. Mason Dunlap, my supervisor, and Bill Howell, a longtime friend, neighbor, and role model, were particularly encouraging and even interested (or at least they did a fabulous job of feigning said interest) in the progress of the book. My debt of gratitude to them is enormous.

Mike and Debbie Hassell, my father and mother, were also supportive and caring. I hope I do justice to their expectations. Thanks also to Aaron and Julie Slyter for their friendship, Tom Syroid for the inspiration to write books, and Robert Bruce Thompson for guidance.

Special thanks to Alan DeKok of the FreeRADIUS project and Niels Jonker of Boingo Wireless for their timely and Herculean efforts to review this book. I believe I owe both of them a couple of beers.

Last but by no means least, my longtime girlfriend, Anna Watson, was by my side through thick and thin and suffered through more than one weekend during which I was focused on email and chapter writing instead of romantic dinners and movie watching. I suspect she will require an approval form signed in triplicate before I write another book. (Who can blame her?) This book would never have gotten off the ground were it not for her support and love.

# Chapter 1. An Overview of RADIUS

In an ideal world, we wouldn't have to use authentication of any type to gain access to anything. But as long as free enterprise exists and access to private resources is sold, authentication will exist.

You may have experienced authentication as recently as an hour ago, when you used a dial-up Internet account to log on and surf the Web for the latest headlines. You may have checked your corporate email on your PalmPilot to see if your biggest client had returned your message about the newest proposal. And this weekend, when you use a VPN to connect to your office network so you can revise that presentation that's due early Monday morning, you'll have to authenticate yourself.

But what goes on behind the scenes when you prove your identity to a computer? After all, the computer has to have a set of processes and protocols to verify that you are indeed who you say you are, find out what you are allowed to access, and finally, tell you all of this. There's one protocol that does this all: the Remote Access Dialin User Service, or RADIUS.

RADIUS, originally developed by Livingston Enterprises, is an access-control protocol that verifies and authenticates users based on the commonly used challenge/response method. (I'll talk more about challenge/response authentication later.) While RADIUS has a prominent place among Internet service providers, it also belongs in any environment where central authentication, regulated authorization, and detailed user accounting is needed or desired.

# 1.1 An Overview of AAA

The framework around which RADIUS is built is known as the AAA process, consisting of authentication, authorization, and accounting. While there's nothing specific to RADIUS in the AAA model, a general background is needed to justify most of RADIUS's behavior. RADIUS was created before the AAA model was developed, but it was the first real AAA-based protocol exhibiting the AAA functionality to earn industry acceptance and widespread use. However, that's not to say there aren't other protocols that satisfy the architecture's requirements.

This model serves to manage and report all transactions from start to finish. The following questions serve well as a mimicking of the functionality by asking:

- Who are you?
- What services am I allowed to give you?
- What did you do with my services while you were using them?

To begin, let's look at why the AAA architecture is a better overall strategy than others. Before AAA was introduced, individual equipment had to be used to authenticate users. Without a formal standard, each machine likely had a different method of authentication—some might have used profiles, while others might have used Challenge/Handshake Authentication Protocol (CHAP) authentication, and still others might have queried a small internal database with SQL. The major problem with this helter-skelter model is one of scalability: while keeping track of users on one piece of network equipment might not be a huge manageability obstacle, increasing capacity by adding other equipment (each with its own authentication methods) quickly ballooned the process into a nightmare. Kludgy scripts were written to halfway automate the process, but there was no real way to monitor usage, automatically authenticate users, and seamlessly provide a variety of services.

The AAA Working Group was formed by the IETF to create a functional architecture that would address the limitations of the system described above. Obviously, there was a need to focus on decentralizing equipment and monitoring usage in heterogeneous networks. ISPs began offering services other than just standard dial-up, including ISDN, xDSL, and cable-modem connectivity, and there needed to be a standard way in which users could be verified, logged on, and monitored throughout the network. After much work, the AAA architecture was born.

## A Word About Terminology

When discussing AAA and RADIUS, the terms "client" and "server" often come up. However, there can be some confusion about which of these roles a particular machine is playing in a specific transaction. Let's take a look at each of these roles.

# 1.2 Key Points About AAA Architecture

The AAA architecture, simply put, is an attempt to map out a design of how the AAA pieces fit together. AAA implementations can be as simple or as complex as they need to be, mainly because of the efforts of the Internet Research Task Force (IRTF) AAA Architecture Working Group to make a model as application-neutral as possible. In other words, the AAA model is designed to work in environments with varied user requirements and equally varied network design. There are some key attributes of the model that make this possible.

First, the AAA model depends on the client/server interaction, in which a client system requests the services or resources of a server system. In simple implementations, these roles generally stick—the server never acts as the client and vice versa. Client/server environments allow for a good load-balancing design, in which high availability and response time are critical. Servers can be distributed and decentralized among the network. Contrast this with the opposite network model, a peer-to-peer (P2P) network. With P2P networks, all systems display characteristics of both client and server systems, which can introduce such demons as processing delays and unavailability.

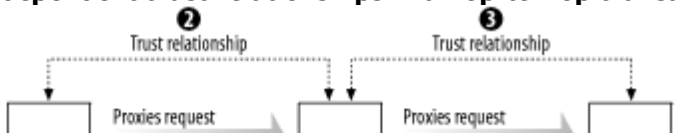
A proxying capability is a slight variation of this. An AAA server can be configured to authorize a request or pass it along to another AAA server, which will then make the appropriate provisions or pass it along again. In essence, a proxy chain is created, in which AAA servers make requests of both clients and other AAA servers. I said "slight variation" earlier because when a server proxies another server, the originator displays the characteristics of a client. Thus, a trust relationship has to be created for each client/server hop until the request reaches equipment that provisions the needed resources.

Proxying is a very useful feature of the AAA model and a boon to enterprise and distributed network implementations, in which some AAA equipment can be configured to always proxy requests to machines in other locations. An example of proxying at its best is with an ISP reseller agreement. Often a major networking company will make a significant investment in network infrastructure and place points of presence in multiple locations. Armed with this distributed network, the company then resells to smaller ISPs that wish to expand their coverage and take advantage of a better network. The reseller has to provide some form of access control over the tangible resources in each location, but the smaller ISP doesn't wish to share personal information about its users with the reseller. In this case, a proxying AAA machine is placed at each of the reseller's points of presence, and those machines then communicate with the appropriate NAS equipment at the smaller ISP.

Clients requesting services and resources from an AAA server (and in this case, clients can include AAA proxies) can communicate with each other by using either a hop-to-hop or an end-to-end transaction. The distinction is where the trust relationship lies in the transaction chain. Consider the following circumstances to get a better picture.

In a hop-to-hop transaction, a client makes an initial request to an AAA device. At this point, there is a trust relationship between the client and the frontline AAA server. That machine determines that the request needs to be forwarded to another server in a different location, so it acts as a proxy and contacts another AAA server. Now the trust relationship is with the two AAA servers, with the frontline machine acting as the client and the second AAA machine acting as the server. It's important to note that the trust relationship is not inherently transitive, meaning that the initial client and the second AAA machine do not have a trust relationship. Figure 1-1 shows how the trusts are sequential and independent of each other.

**Figure 1-1. Independent trust relationships in a hop-to-hop transaction**



# 1.3 The Authorization Framework

Moving on in the soup of terminology, we come to the AAA Authorization Framework, an RFC document from the subset of the AAA Working Group set up by the IETF. Like an architecture document, a framework is designed as a roadmap, but it tends to be a bit more specific. Frameworks designate how systems interact with one another, but frameworks generally concentrate more on models specific to certain environments, such as an Internet wholesaler, a corporate VPN center, or other similar situations.

First, though, we should point out the distinctions in terminology. The authorization framework introduces the concept of a User Home Organization (UHO), which is an entity that has a direct contractual relationship with an end user. Also, the Service Provider (SP) is involved, which maintains and provisions the tangible network resources. The UHO and the SP need not be the same organization; a good example of this is, again, an ISP wholesaler or reseller that provides its own network resources to other organizations. For the purposes of this overview, I'll first look at scenarios in which the UHO and SP are one and the same, and then I'll cover a more detailed scenario that is commonly found.

## 1.3.1 Authorization Sequences

There are several different methods in which the end user, the AAA server, and the network equipment communicate during a transaction. Specifically, there are three different sequences in which each machine is contacted.

*The agent sequence*

In this sequence, the AAA server acts as a middleman of sorts between the service equipment and the end user. The end user initially contacts the AAA server, which authorizes the user's request and sends a message to the service equipment notifying it to set that service up. The service equipment does so, notifies the AAA machine, and the notification is passed on to the end user, who then begins using the network. This sequence is typically used in broadband applications in which quality of service (QoS) is part of an existing contract.

*The pull sequence*

Dial-in users frequently encounter this sequence. The end user in this situation connects directly to the service equipment (terminal gear or other NAS machinery), which then checks with an AAA server to determine whether to grant the request. The AAA server notifies the service equipment of its decision, and the service equipment then either connects or disconnects the user to the network.

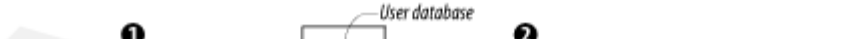
*The push sequence*

The push sequence alters the trust relationship between all of the machines in a transaction. The user connects to the AAA server first, and when the request to the server is authorized, the AAA server distributes some sort of authentication "receipt" (a digital certificate or signed token, perhaps) back to the end user. The end user then pushes this token along with his request to the service equipment, and the equipment treats the ticket from the AAA server as a green light to provision the service. The main distinction is that the user acts as the agent between the AAA server and the service equipment.

Here are some diagrams of the sequences that visually indicate the authorization transaction sequence.

Figure 1-3 shows the agent sequence, in which an AAA server acts as the middleman between the client and the service equipment responsible for provisioning the client's request.

**Figure 1-3. The agent sequence**



# 1.4 And Now, RADIUS

There's been much talk of AAA and so little of RADIUS. This is largely because RADIUS will not be eternally the access control protocol of choice. In fact, RADIUS was created by a separate working group long before the AAA design and fundamentals were brought to existence. The similarities are, however, remarkable.

AAA is the foundation of the next generation remote access protocol. Developments in creating the next protocol are being made as I write this, so the days of RADIUS being the standard aren't infinite. But on the same token, RADIUS has an established and well-respected presence in the industry, so it has a definite future.

## 1.4.1 A Brief History

RADIUS, like most innovative products, was built from a need. In this case, the need was to have a method of authenticating, authorizing, and accounting for users needing access to heterogeneous computing resources. Merit Networks, a big player in creating the Internet as we know it, operated a pool of dial-up resources across California. At the time, authentication methods were peculiar to specific pieces of equipment, which added a lot of overhead and didn't allow for much in the way of management flexibility and reporting. As the dial-up user group grew, the corporation realized they needed a mechanism more flexible and extensible than remaining with their proprietary, unwieldy equipment and scripts. Merit sent out a request for proposal, and Livingston Enterprises was one of the first respondents. Representatives for Merit and Livingston contacted each other, and after meeting at a conference, a very early version of RADIUS was written. More software was constructed to operate between the service equipment Livingston manufactured and the RADIUS server at Merit, which was operating with Unix. The developer of RADIUS, Steve Willins, still remains on the RFC document. From that point on, Livingston Enterprises became Lucent, and Merit and Lucent took the RADIUS protocol through the steps to formalization and industry acceptance. Both companies now offer a RADIUS server to the public at no charge.

## 1.4.2 Properties of RADIUS

The RFC specifications for the RADIUS protocol dictate that RADIUS:

- Is a UDP-based connectionless protocol that doesn't use direct connections
- Uses a hop-by-hop security model
- Is stateless (more to come on that later)
- Supports PAP and CHAP authentication via PPP
- Uses MD5 for password-hiding algorithms

# Chapter 2. RADIUS Specifics

In this chapter, I'll step through the most important sections of the RADIUS RFC and interpret them. Since the RFC is approximately 80 pages long, it's not appropriate to provide every detail here. Some portions of the document are antiquated, seldom used, or simply not important. While formality dictates their presence in the official document, this chapter is meant more as a working reference guide.

# 2.1 Using UDP versus TCP

A question frequently asked of the RADIUS development team is why the protocol uses the UDP protocol instead of TCP. For purely operational requirements, UDP was selected largely because RADIUS has a few inherent properties that are characteristic of UDP: RADIUS requires that failed queries to a primary authentication server be redirected to a secondary server, and to do this, a copy of the original request must exist above the transport layer of the OSI model. This, in effect, mandates the use of retransmission timers.

The protocol bets on the patience of users to wait for a response. It assumes some middle ground between lightning fast and slow as molasses. The RADIUS RFC describes it best: "At one extreme, RADIUS does not require a "responsive" detection of lost data. The user is willing to wait several seconds for the authentication to complete. The generally aggressive TCP retransmission (based on average round trip time) is not required, nor is the acknowledgment overhead of TCP. At the other extreme, the user is not willing to wait several minutes for authentication. Therefore the reliable delivery of TCP data two minutes later is not useful. The faster use of an alternate server allows the user to gain access before giving up."

Since RADIUS is stateless (as I mentioned in Chapter 1), UDP seems natural, as UDP is stateless, too. With TCP, clients and servers must have special code or administrative workarounds to mitigate the effects of power losses, reboots, heavy network traffic, and decommissioning of systems. UDP prevents this headache since it allows one session to open and remain open throughout the entire transaction.

To allow for heavy systems use and traffic on the backend, which can sometimes delay queries and look-ups by as much as 30 seconds or more, it was determined that RADIUS should be multithreaded. UDP allows RADIUS to spawn to serve multiple requests at a time, and each session has full, uninhibited communication abilities between the network gear and the client. Thus, UDP was a good fit.

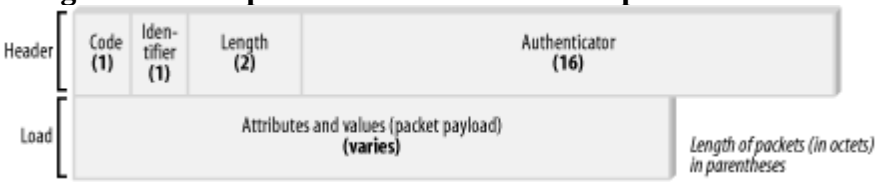
The only downside to using UDP is that developers must create and manage retransmission timers themselves—this capability is built into TCP. However, the RADIUS group felt that this one downside was far outweighed by the convenience and simplicity of using UDP. And so it was.

# 2.2 Packet Formats

The RADIUS protocol uses UDP packets to pass transmissions between the client and server. The protocol communicates on port 1812, which is a change from the original RADIUS RFC document. The first revision specified that RADIUS communications were to take place on port 1645, but later this was found to conflict with the "Datametrics" service.

RADIUS uses a predictable packet structure to communicate, which is shown in Figure 2-1.

**Figure 2-1. A depiction of the RADIUS data packet structure**




The data structure is broken down into five distinct regions, which are discussed later in this chapter.

## 2.2.1 Code

The code region is one octet long and serves to distinguish the type of RADIUS message being sent in that packet. Packets with invalid code fields are thrown away without notification. Valid codes are:
*1*

Access-Request
*2*

Access-Accept
*3*

Access-Reject
*4*

Accounting-Request
*5*

Accounting-Response
*11*

Access-Challenge
*12*

Status-Server (under continued development)
*13*

Status-Client (under continued development)
*255*

Reserved

## 2.2.2 Identifier

# 2.3 Packet Types

At this point, we have covered the structure of the packets RADIUS uses to transmit data. But what do these packets do? There are four RADIUS packet types that are relevant to the authentication and authorization phases of the AAA transaction:
Access-Request Access-Accept Access-Reject Access-Challenge

While the accounting packet types are covered in detail in Chapter 4, the next section will step through these packets and detail their intent, format, and structure.

***Access-Request***

---

| Packet Type | Response |
|---|---|
| Code | 1 |
| Identifier | Unique per request |
| Length | Header length plus all additional attribute data |
| Authenticator | Request |
| Attribute Data | 2 or more |

The Access-Request packet is used by the service consumer when it is requesting a particular service from a network. The client sends a Request packet to the RADIUS server with a list of the requested services. The key factor in this transmission is the code field in the packet header: it must be set to 1, the unique value of the Request packet. The RFC states that replies must be sent to all valid Request packets, whether the reply is an authorization or a rejection.

The payload of the Access-Request packet should include the username attribute to identify the person attempting to gain access to the network resource. The payload is required to contain the IP address or canonical name of the network equipment from which it is requesting service. It also has to contain a user password, a CHAP-based password, or a state identifier, but not both types of passwords. The user password must be hashed using MD5.

How do these rules apply to RADIUS proxy chains? Basically, new packets need to be created whenever attributes are changed, since identifying information is changed. Attributes with shared secrets, which are covered in detail later in this chapter, need to be reversed by the proxy server (to obtain the original payload information) and then encrypted again with the secret that the proxy server shares with the remote server.

# 2.4 Shared Secrets

To strengthen security and increase transactional integrity, the RADIUS protocol uses the concept of shared secrets. Shared secrets are values generated at random that are known to both the client and the server (hence the "shared"). The shared secret is used within all operations that require hiding data and concealing values. The only technical limitation is that shared secrets must be greater than 0 in length, but the RFC recommends that the secret be at least 16 octets. A secret of that length is virtually impossible to crack with brute force. The same set of best practices that dictate password usage also govern the proper use of RADIUS shared secrets.

Shared secrets (commonly called just "secrets") are unique to a particular RADIUS client and server pair. For instance, if an end user subscribes to multiple Internet service providers for his dial-up access, he indirectly makes requests to multiple RADIUS servers. The shared secrets between the client NAS equipment in ISPs A, B, and C that are used to communicate with the respective RADIUS servers should not match.

While some larger scale RADIUS implementations may believe that protecting transactional security by using an automated shared-secret changer is a prudent move, there is a rather large pitfall: there is no guarantee the clients and servers can synchronize to the new shared secret at the most appropriate time. And even if it was certain that the simultaneous synchronization could occur, if there are outstanding requests to the RADIUS server and the client is busy processing (and, therefore, it misses the cue to synchronize the new secret), then those outstanding requests will be rejected by the server. The situation would be tantamount to having your checking account numbers stolen: when the bank gives you new account numbers, outstanding checks written on your old account will bounce since that account was closed.

# 2.5 Attributes and Values

Although at this point the attribute field in a RADIUS packet may seem like nothing more than a glorified way to determine header information, there's a lot more going on than meets the eye. Specifically, the entire RADIUS transaction is built around passing to and from the client and server attribute-value pairs (AVPs) that contain virtually every property and characteristic of the AAA transaction.

To enhance security, the RADIUS RFC restricts some attributes from being sent in certain packets—or to be more specific, the timing of certain packets. For instance, to prevent the password from ever crossing the wire more than once for one authentication/authorization process, the User-Password attribute is never allowed to be sent in a reply packet from the server to the client. Even more stringently, the RFC prevents some attributes from even being present in certain transactions, while others can appear more than once, and still others only once. More information on restrictions like these is presented in the sections that follow.

Attributes in a packet all follow a specific field format. From this point on, I'll refer to this field format as:
*Attribute Number*

This number denotes the type of attribute presented in the packet. The attribute's name is not passed in the packet—just the number. Generally, attribute numbers can range from 1-255, with one specific number serving as a "gateway" of sorts for vendors to provide their own specific attributes.
*Attribute Length*

This field describes the length of the attribute field, which must be three or greater. It behaves in much the same way as the length field of the RADIUS packet header.
*Value*

Containing the property or characteristic of the attribute itself, this field is required for each attribute presented, even if the value itself is null. The length of this will vary based on the inherent nature of the attribute itself.

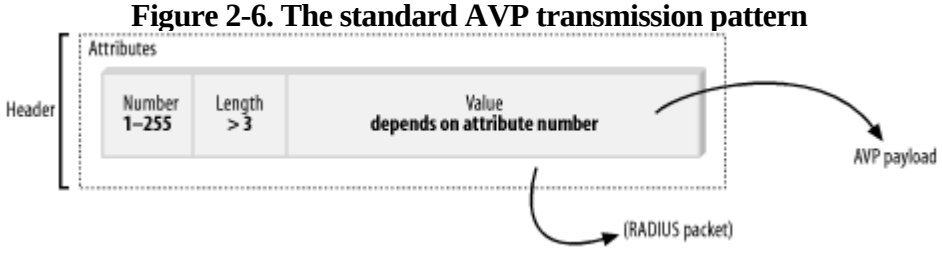
The concepts of attributes and values themselves are worthy of a bit more discussion.

## 2.5.1 Attributes

Attributes simply describe a behavior or a property of a type of service. While most attributes are included to denote a particular setting for a service type, the presence of some attributes in the packet tells the RADIUS server what it needs to know. As you'll see later in this chapter, the very inclusion of the CHAP-Password attribute in a packet signals to the RADIUS server the proper hashings and password-concealing processes to perform for that particular transaction. This is a unique property of attributes—they can stand alone, while values simply cannot.

Attributes are transmitted inside the RADIUS packet in a predetermined, standard format, as shown in Figure 2-6.

**Figure 2-6. The standard AVP transmission pattern**



# 2.6 Authentication Methods

RADIUS supports a variety of different protocol mechanisms to transmit sensitive user-specific data to and from the authentication server. The two most common are the Password Authentication Protocol (PAP) and the CHAP. RADIUS also allows for more attributes and methods developed by vendors, including support for features peculiar to Windows NT, Windows 2000, and other popular network operating systems and directory services. The following section explores the two most common methods in greater detail.

## 2.6.1 PAP

The User-Password attribute in a requesting packet signals to the RADIUS server that the PAP protocol will be used for that transaction. It's important to note that the only mandatory field in this case is the User-Password field. The User-Name field does not have to be included in the requesting packet, and it's entirely possible that a RADIUS server along a proxy chain will change the value in the User-Name field.

The algorithm used to hide the original user's password is composed of many elements. First, the client detects the identifier and the shared secret for the original request and submits it to an MD5 hashing sequence. The client's original password is put through the XOR process and the result coming from these two sequences is then put in the User-Password field. The receiving RADIUS server then reverses these procedures to determine whether to authorize the connection. The very nature of the password-hiding mechanism prevents a user from determining if, when the authentication fails, the failure was caused by an incorrect password or an invalid secret. Most commercial RADIUS servers, though, include logic that looks at the series of packets previously transmitted from the same client. If a number passes through the connection correctly, most likely the few packets that failed did so because of an incorrect password.

## 2.6.2 CHAP

CHAP is based on the premise that the password should never be sent in any packet across a network. CHAP dynamically encrypts the requesting user's ID and password. The user's machine then goes through its logon procedure, having obtained a key from the RADIUS client equipment of at least 16 octets in length. The client then hashes that key and sends back a CHAP ID, a CHAP response, and the username to the RADIUS client. The RADIUS client, having received all of the above, places the CHAP ID field into the appropriate places in the CHAP-Password attribute and then sends a response. The challenge value originally obtained is placed in either the CHAP-Challenge attribute or in the authenticator field in the header—this is so the server can easily access the value in order to authenticate the user.

To authenticate the user, the RADIUS server uses the CHAP-Challenge value, the CHAP ID, and the password on record for that particular user and submits it to another MD5 hashing algorithm. The result of this algorithm should be identical to the value found in the CHAP-Password attribute. If it's not, the server must deny the request; otherwise, the request is granted.

The fact that the password in a CHAP transaction is never passed across the network is just one reason why CHAP is an appealing authentication protocol. How does this work? The user data against which the hashing routine is run returns a one-way value that does not contain the password. So the server must have the current user's password stored in clear text in its own records in order to create a hash with which to compare. CHAP IDs are also non-persistent, which reduces the possibility of a third party sniffing or otherwise intruding on the transaction. Additionally, the CHAP protocol supports challenging the client anytime during the user's session, which increases the likelihood that invalid users are kept out of the system.

# 2.7 Realms

While RADIUS can be as ignorant of externalities as an administrator wants, it can also be made aware of various implementations. RADIUS is flexible with regard to various design schemes to allow it to support different business and infrastructure models. Take, for instance, a cooperative agreement among three regional Internet service providers. Let's explore this example in greater detail.

Northwest Internet serves the northern and western portions of a state. Southeast Internet serves the southern and eastern regions, and Central State Internet provides support to the central area of a state. While each of these ISPs may have modem-pool resources in overlapping geographical areas, most of the access resources are confined to particular regions.

Now, each of the service providers determine that there is sufficient demand to offer a roaming service to customers to allow them to dial a local number anywhere in the state to access the Internet. While the service would be more expensive than normal, with a home-area dial-up service, a local number allows the customer to avoid expensive long-distance charges most hotels and other lodging establishments levy. Each ISP determines that it's not fiscally efficient for them to construct points of presence in each region, so they form a cooperative alliance in which each ISP allows the other two ISPs to have access to their respective modem pools. So Northwest Internet can offer a roaming service to its mobile users who happen to dial up in the southern and eastern portions of the state, and so on.

The key question here revolves around how each ISP can offer access and ensure that only valid users can connect to their resources, while protecting the sanctity and security of the respective providers' sensitive customer information. To fill this need, RADIUS comes with support for identifying users based on discrete design-based areas, or realms. Realms are identifiers that are placed before or after the values normally contained in the User-Name attribute that a RADIUS server can use to identify which server to contact to start the AAA process.

The first type of realm identifier is known as the prefix realm, in which the realm name is placed before the username, and the two are separated by a preconfigured character, most commonly @, \, or /. For instance, a user named *jhassell* who subscribes to Central State Internet's service (whose realm name is CSI) would configure his client to pass a username like *CSI\jhassell*.

The other realm identifier syntax is the suffix realm, where the username is placed before the realm name. The common separators are still used in this syntax as well, though by far the most common is the @ sign. For example, the user *awatson* subscribing to Northwest Internet's service (realm name: NWI) using realm suffix identification would pass a username like *awatson@NWI*.

# 2.8 RADIUS Hints

An administrator can configure a RADIUS server to grant some services by default to any authenticated user, while other configurations might permit only the services requested in the client's request packet to be authorized. RADIUS can be set up to handle service authorizations in countless different ways. The RADIUS RFC thus specifies information that can be included in a RADIUS packet header sent from a client to a server that "hints" to the server which explicit services it wants. These bits of information are called RADIUS hints.

RADIUS hints behave differently based on the way an administrator sets up his RADIUS client gear to authorize transactions. The RFC states that the receiving RADIUS server can choose whether to grant the hints requests if doing so would not violate the local security setup. If the RADIUS server chooses not to grant the hints request, though, it is also allowed under the RFC specification to authorize a service that can be granted based on the user's access policy. If it can't do this, then it must terminate and disconnect the session.

Hints are designed primarily for environments in which the RADIUS server has partial control of the resources needed to provision service for the client. For instance, the client may request a specific, static IP as paid for in her monthly billing by sending a hint in the request. The NAS gear, having obtained explicit authorization from the RADIUS server (eliminating the extra transaction hop to obtain authorization from the IP leasing pool machine), may then grant the request by telling the RADIUS server to send the details in an Access-Accept packet, alter the routing tables, and do whatever else needs done to provision the service.

It's important to note that RADIUS hints never have any effect on the base RADIUS protocol. They're simply small notes "under the table" to the RADIUS server from the client, requesting that the service have optional, temporary, or extra characteristics or abilities .

# Chapter 3. Standard RADIUS Attributes

In this chapter, I'll look at the global set of standard RADIUS attributes as per the RADIUS RFC. There are 63 attributes defined in the RFC that provide support and configuration options for everything from connection type, virtual terminals, and connect/session time limits to packet filtering and caller-return services. This chapter presents these attributes in alphabetical order.

One note: this chapter covers only the attributes based on the authentication and authorization processes of a RADIUS transaction, which are attributes 1-39 and 60-63. Attributes 40-59 are covered in Chapter 4.

# 3.1 Attribute Properties

Each attribute in this chapter is presented as a separate "nugget" of information. Each nugget contains a quick-reference chart for the particulars of the attribute, followed by a discussion of the attribute, where I discuss any special considerations in the usage or configuration of the attribute, how its use affects or requires other attributes, practical applications of the attribute, and how it sometimes differs from the theoretical implication from the RFC.

Appendix A contains a chart with all of the global standard RADIUS attributes (including those specific to accounting) and their numbers, lengths, values, and packet presence requirements.

Chapter 9 presents the attributes introduced and revised in the new RADIUS Extensions RFCs. I have separated these attributes to maintain the distinction exhibited in the RFCs.

***Callback-ID***

---

| | |
|---|---|
| Attribute Number | 20 |
| Length | 3 or more octets |
| Value | STRING |
| Allowed in | Access-Accept |
| Prohibited in | Access-Request, Access-Reject, Access-Challenge |
| Presence in Packet | Not required |
| Maximum Iterations | 1 |

This attribute is used when a RADIUS implementation is set up to return a user's call. This is commonly used in corporate situations to avoid long-distance charges in hotel rooms and other remote locations. This value, a STRING, is often the identifier for a profile configured on the service equipment; there is no specific standard for a string name, a triggered action, or something else. In other words, it is environment-specific. RADIUS client gear is allowed to reject a connection if this attribute is present but not supported by that gear.

***Callback-Number***

# Chapter 4. RADIUS Accounting

ISPs often manage points of presence over several locations, most likely geographically dispersed. All of these points of presence require protection to guard against unauthorized use of the expensive network to which they allow access. Although the front line of defense may (and should) be a robust and extensible form of authentication (to verify a user's declared identity) and authorization (to provide a user with only the services to which he is entitled), much valuable information can be gleaned from data collected about users' activities on the network. Which user logged on? When did she do so? What services was he granted?

The data becomes even more useful when it is compiled to analyze a group of users. What is the average call time for a user? How much data does that user transfer? Do I, as a system administrator, need to set a time limit for a single session so as to protect limited dial-in resources? Do I have users that are abusing an on-demand connection? All of these questions can be answered using information mined from the accounting process.

RADIUS supports a full-featured accounting protocol subset, which allows it to satisfy all requirements of the AAA model. This chapter describes the design, operation, packets, and attributes that are specific and germane to RADIUS accounting.

# 4.1 Key Points in RADIUS Accounting

The design of accounting in RADIUS is based upon three major characteristics:
*Accounting will be based on a client/server model.*

The RADIUS accounting machine is the server to the RADIUS client gear, which acts as the client. The client passes the usage data to the RADIUS server for processing. The RADIUS server acknowledges successful receipt of the data. It is also possible for the RADIUS server to act as an accounting proxy, much like the similar capability in the authentication and authorization realms.
*Communications between devices will be secure.*

All data is passed to and from the RADIUS server and the client gear through the use of a shared secret, which is never transmitted across the wire.
*RADIUS accounting will be extensible.*

The format of the accounting attributes is much like those of the authentication and authorization attributes, in that most of the services offered by the implementations can be defined and qualified using AVPs. AVPs can be added and modified to an existing implementation without disrupting the functionality already in use.

# 4.2 Basic Operation

All communications regarding RADIUS accounting are done with an Accounting-Request packet. A client that is participating in the RADIUS accounting process will generate an Accounting Start packet, which is a specific kind of Accounting-Request packet. This packet includes information on which service has been provisioned and on the user for which these services are provided. This packet is sent to the RADIUS accounting server, which will then acknowledge receipt of the data. When the client is finished with the network services, it will send to the accounting server an Accounting Stop packet (again, a specialized Accounting-Request packet), which will include the service delivered; usage statistics such as time elapsed, amount transferred, average speed; and other details. The accounting server acknowledges receipt of the stop packet, and all is well. If the server does not or cannot handle the contents of the Accounting-Request packet, it is not allowed to send a receipt acknowledgment to the client.

In this instance, the RFC recommends that a client continue to send its packets to the accounting server when it has not received an acknowledgment that its Accounting-Request packet has been processed. In fact, in large distributed networks, it is desirable to have several accounting servers act in a round-robin fashion to handle failover and redundancy needs. An administrator can carry this mentality further and designate certain accounting servers to handle different requests—one for his dial-up users, one for his DSL customers, and yet another for ISDN connections. Additionally, the proxy functionality present in the authentication and authorization realms of RADIUS are also allowed in the accounting phase, as the accounting server may make requests of other servers to assist in the processing of Accounting-Request packets.

## 4.2.1 More on Proxying

RADIUS accounting proxies act in much the same way as RADIUS authentication/authorization proxies do. Consider the following process:

1. The RADIUS client gear sends the Accounting Start packet to the accounting server.
2. The receiving accounting server logs the packet. It may then add the Proxy-State attribute and accompanying details (though it is not required to do so). It updates the request authenticator and then forwards the information to a remote machine.
3. This remote machine logs the incoming, forwarded packet. It then does what the first server could not do (that is to say, it performs the action that was required of the proxy), retains and copies all of the Proxy-State attributes exactly as they appeared, and sends an Accounting-Response packet back to the original forwarding server.
4. The original forwarding server receives the acknowledgment, strips out the Proxy-State information, constructs and adds the Response Authenticator, and sends the modified acknowledgment response back to the RADIUS client gear.

Figure 4-1 shows the flow of this process.

# 4.3 The Accounting Packet Format

As mentioned in Chapter 2, the RADIUS protocol uses a UDP foundation to transmit packets between clients, servers, and proxies. While the original RADIUS accounting RFC (number 2139, to be exact) specified that accounting transactions should take place on port 1646, the latest RFC (2866) changed the port to 1813, because port 1646 was already assigned to the sa-msg-port service.

The packets are broken down into four distinct regions, which are discussed next.

## 4.3.1 Code

The code region is one-octet long and indicates the type of RADIUS accounting information transmitted in that packet. Packets with invalid code fields are thrown away without notification. Valid codes are:

*4*

Accounting-Request

*5*

Accounting-Response

## 4.3.2 Identifier

The identifier region is one-octet long and is used to perform threading, or the automated linking of initial requests and subsequent replies. RADIUS accounting servers can generally intercept duplicate messages by examining such factors as the source IP address, the source UDP port, the time span between the suspect messages, and the identifier field.

## 4.3.3 Length

The length region is two-octets long and is used to specify the length of a RADIUS accounting message. The value in this field is calculated by analyzing the code, identifier, length, authenticator, and attribute fields and finding their sum. The length field is checked to ensure data integrity when an accounting server receives a packet. Valid length values range between 20 and 4095.

The RFC specification requires certain behaviors of RADIUS servers with regard to incorrect length data. If the accounting server receives a transmission with a message longer than the length field, it ignores all data past the end point designated in the length field. Conversely, if the server receives a shorter message than the length field reports, the server will discard the message.

## 4.3.4 Authenticator

The authenticator region, often 16-octets long, is the field in which the integrity of the packet's payload is inspected and verified. In this field, the most important octet—the value used to authenticate replies from the accounting server—is transmitted before any other.

There are two distinct types of authenticators: the request and response authenticators. Request authenticators,

# 4.4 Accounting Packet Types

At this point, we have covered the structure of the packets that RADIUS uses to transmit accounting data. But we need to establish the identity and properties of these specific packets. There are two RADIUS packet types that are relevant to the accounting phase of an AAA transaction:

- Accounting-Request
- Accounting-Response

The next section will step through these packets and detail their intent, format, and structure.

### *Accounting-Request*

| | |
|---|---|
| Packet Type | Request |
| Code | 4 |
| Identifier | Unique for each request; unique for each transmission of modified data |
| Authenticator | Request |
| Attribute Data | 0 or more attributes |

Accounting-Request packets are sent from the client to the server. Remember that a client can be a true RADIUS client or another RADIUS server acting as a proxy. The client sends the packet with the code field set to 4. When the server receives this request packet, it is required to transmit an acknowledgment to the client unless it cannot handle or process the packet. In this case, it cannot transmit anything to the client.

With the exceptions of the User-Password, CHAP-Password, Reply-Message, and State attributes, any other attribute allowed in an Access-Request or Access-Accept packet can be used inside an Accounting-Request packet.

Chapter 3 discusses all standard RADIUS attributes and their properties, including the packets in which they are allowed to be included. Check there for a complete overview of

# 4.5 Accounting-specific Attributes

In the following section, I'll cover the attributes of the global RADIUS space that are specific to the accounting phase of an AAA transaction. Much like in Chapter 3, each of the current 12 accounting-specific attributes will be a separate tidbit of information, including an at-a-glance properties chart and a short discussion of key points and important considerations. Again, Appendix A is a chart of the entire global RADIUS attribute list, covering all phases of the AAA model, and should serve as a useful quick reference.

### *Acct-Status-Type*

| | |
|---|---|
| Attribute Number | 40 |
| Length | 6 |
| Value | ENUM |
| Allowed in | Accounting-Request |
| Prohibited in | Accounting-Response |
| Presence in Packet | Required |
| Maximum Iterations | 1 |

This attribute indicates whether the Accounting-Request packet is being sent upon the user first authenticating and connecting to the network or upon the user finishing use of the services and disconnecting. It can also be used to mark when to start and stop accounting should the RADIUS client gear require rebooting or other system maintenance. Note that when RADIUS client gear crashes, stop records in general are not sent to the accounting server. Obviously, this has the potential to mess up accounting data, and a crashed client is not all that uncommon.

The payload value of the attribute contains 15 possible values, each of which are listed in Table 4-1.

Table 4-1. Values for the Acct-Status-Type attribute

| **Value** | **Status type** |
|---|---|
| 1 | Start |

# Chapter 5. Getting Started with FreeRADIUS

Up to this point, I've talked about the theoretical underpinnings of both the authentication-authorization-accounting (AAA) architecture as well as the specific implementation of AAA characteristics that is the RADIUS protocol. I will now focus on practical applications of RADIUS: implementing it, customizing it for your specific needs, and extending its capabilities to meet other needs in your business. First, though, I need a product that talks RADIUS.

Enter FreeRADIUS.

# 5.1 Introduction to FreeRADIUS

The developers of FreeRADIUS speak on their product and its development, from the FreeRADIUS web site:

FreeRADIUS is one of the most modular and featureful [sic] RADIUS servers available today. It has been written by a team of developers who have more than a decade of collective experience in implementing and deploying RADIUS software, in software engineering, and in Unix package management. The product is the result of synergy between many of the best-known names in free software-based RADIUS implementations, including several developers of the Debian GNU/Linux operating system, and is distributed under the GNU GPL (version 2).

FreeRADIUS is a complete rewrite, ground-up compilation of a RADIUS server. The configuration files exhibit many similarities to the old Livingston RADIUS server. The product includes support for:

- Limiting the maximum number of simultaneous logons, even on a per-user basis
- More than one DEFAULT entry, with each being capable of "falling through" to the next
- Permitting and denying access to users based on the *huntgroup* to which they are connected
- Setting certain parameters to be huntgroup specific
- Intelligent "hints" files that select authentication protocols based on the syntax of the username
- Executing external programs upon successful login
- Using the $INCLUDE filename format with configuration, users, and dictionary files
- Vendor-specific attributes
- Acting as a proxy RADIUS server

FreeRADIUS supports the following popular NAS equipment:

# 5.2 Installing FreeRADIUS

At present, the FreeRADIUS team doesn't offer precompiled binaries. The best way to start off is to grab the latest source code, compressed using tar and *gzip*, from the FreeRADIUS web site at http://www.freeradius.org. Once the file is on your computer, execute the following command to uncompress the file:

```
tar -zxvf freeradius.tar.gz
```

Next, you'll need to compile FreeRADIUS. Make sure your system at least has *gcc*, *glibc*, *binutils*, and *gmake* installed before trying to compile. To begin compiling, change to the directory where your uncompressed source code lies and execute *./configure* from the command line. You can also run *./configure -flags* and customize the settings for the flags in Table 5-1.

Table 5-1. Optional configuration flags for FreeRADIUS

| **Flag** | **Purpose** | **Default** |
|---|---|---|
| `--enable-shared[=PKGS]` | Builds shared libraries. | Yes |
| `--enable-static[=PKGS]` | Builds static libraries. | Yes |
| `--enable-fast-install[=PKGS]` | Optimizes the resulting files for fastest installation. | Yes |
| `--with-gnu-ld` | Makes the procedure assume the C compiler uses GNU lD. | No |
| `--disable-libtool-lock` | Avoids locking problems. This may break parallel builds. | Not applicable |
| `--with-logdir=DIR` | Specifies the directory for log files. | `LOCALSTATEDIR/log` |
| `--with-radacctdir=DIR` | Specifies the directory for detail files. | `LOGDIR/radacct` |
| `--with-raddbdir=DIR` | Specifies the directory for configuration files. | `SYSCONFDIR/raddb` |
| `--with-dict-nocase` | Makes the dictionary case insensitive. | Yes |

# 5.3 In-depth Configuration

At this point, you've compiled, installed, configured, started, and tested a simple FreeRADIUS implementation that is functional. However, 99.5% of the RADIUS/AAA implementations around the world are just not that simple. In this section, I'll delve into the two major configuration files and discuss how to tweak, tune, customize, and effect change to the default FreeRADIUS installation. In Chapter 6, I'll discuss advanced topics, such as pluggable authentication module (PAM) support, integration with MySQL, LDAP usage, and other topics.

## 5.3.1 Configuring radiusd.conf

*radiusd.conf* file is the central location to configure most aspects of the FreeRADIUS product. It includes configuration directives as well as pointers and two other configuration files that may be located elsewhere on the machine. There are also general configuration options for the multitude of modules available now and in the future for FreeRADIUS. The modules can request generic options, and FreeRADIUS will pass those defined options to the module through its API.

Before we begin, some explanation is needed of the operators used in the statements and directives found in these configuration files. The = operator, as you might imagine, sets the value of an attribute. The := operator sets the value of an attribute and overwrites any previous value that was set for that attribute. The == operator compares a state with a set value. It's critical to understand how these operators work in order to obtain your desired configuration.

In this chapter, I'll look at several of the general configuration options inside *radiusd.conf*. Some of the more advanced directives in this file will be covered in Chapter 6.

***pidfile***

---

This file contains the process identification number for the *radiusd* daemon. You can use this file from the command line to perform any action to a running instance of FreeRADIUS. For example, to shut FreeRADIUS down without any protests, issue:

```
kill -9 `cat /var/run/radiusd.pid'
```

**Usage:**

```
pidfile = [path]
```

**Suggestion:**

```
pidfile = ${run_dir}/radiusd.pid
```

***user and group***

---

These options dictate under what user and group *radiusd* runs. It is not prudent to allow FreeRADIUS to run under a user and group with excessive permissions. In fact, to minimize the permissions granted to FreeRADIUS, use the user and group "nobody." However, on systems configured to use shadow passwords, you may need to set the user to "nobody" and the group to "shadow" so that *radiusd* can read the *shadow* file. This is not a desirable idea. On some systems, you may need to set both the user and group to "root," although it's clear why that is an even worse idea.

**Usage:**

```
user = [username]; group = [groupname]
```

**Suggestion:**

# 5.4 Troubleshooting Common Problems

In this section, I'll take a look at some of the most frequently occurring problems with a new FreeRADIUS setup and how to fix them.

## 5.4.1 Linking Errors When Starting FreeRADIUS

If you receive an error similar to the following:

```
 Module: Loaded SQL
rlm_sql: Could not link driver rlm_sql_mysql: file not found
rlm_sql: Make sure it (and all its depend libraries!) are in the search path
radiusd.conf[50]: sql: Module instantiation failed.
```

It means that some shared libraries on the server are not available. There are a couple of possible causes from this.

First, the libraries that are needed by the module listed in the error messages couldn't be found when FreeRADIUS was being compiled. However, if a static version of the module was available, it was built at compile time. This would have been indicated with very prominent messages at compile time.

The other cause is that the dynamic linker on your server is not configured correctly. This would result in the libraries that are required being found at compile time, but not run time. FreeRADIUS makes use of standard calls to link to these shared libraries, so if these calls fail, the system is misconfigured. This can be fixed by telling the linker where these libraries are on your system, which can be done in one of the following ways:

- Write a script that starts FreeRADIUS and includes the variable *LD_LIBRARY_PATH*. This sets the paths where these libraries can be found.
- If your system allows it, edit the /etc/ld.so.conf file and add the directory containing the shared libraries to the list.
- Set the path to these libraries inside radiusd.conf using the *libdir* configuration directive. The radiusd.conf file has more details on this.

## 5.4.2 Incoming Request Passwords Are Gibberish

Gibberish is usually indicative of an incorrectly formed or mismatched shared secret, the phrase shared between the server and the RADIUS client machine and used to perform secure encryption on packets. To identify the problem, run the server in debugging mode, as described previously. The first password printed to the console screen will be inside a RADIUS attribute (e.g., Password = "rneis\dfkjdf7482odf") and the second will be in a logged message (e.g., Login failed [rneis/dfkjdf7482odf]). If the data after the slash is gibberish—ensure it's not just a really secure password—then the shared secret is not consistent between the server and the RADIUS client. This may even be due to hidden characters, so to be completely sure both are the same, delete and re-enter the secret on both machines.

# Chapter 6. Advanced FreeRADIUS

Congratulations! Chances are that, by now, you have a base FreeRADIUS system up, running, and tested to be working correctly. But it's probably not an optimal system for your implementation and needs. In this chapter, I'll take a look at some of the more advanced tools and methods you can use to extend the capabilities of FreeRADIUS and better integrate it with your existing environment.

# 6.1 Using PAM

FreeRADIUS supports the pluggable authentication model, or PAM, but that must be enabled at compile time. (A discussion of PAM is beyond the scope of this book; however, an excellent introduction to PAM, with answers to some frequently asked questions, is available at http://www.kernel.org/pub/linux/libs/pam/FAQ.) However, the current support for PAM is rather non-standard. In most RADIUS distributions, to enable PAM in transactions, enter User-Password = PAM in the *users* file; this is not supported in FreeRADIUS. You must instead use Auth-Type = Pam. For example, here is a configuration stanza for a non-specific (that is to say, default) user configured for PAM authentication, when he logs in from a specific RADIUS client machine:

```
DEFAULT Auth-Type := Pam, NAS-IP-Address == 206.229.254.5
    Service-Type = Framed-User,
    Framed-Protocol = PPP,
    Framed-IP-Address = 255.255.255.254,
    Filter-Id = "20modun",
    Framed-MTU = 1500,
    Framed-Compression = Van-Jacobson-TCP-IP
```

In some configurations, you may have specific entries configured in the */etc/pam.d* file. The following *users* file configuration stanza uses a unique "Pam-Auth = x" identifier to direct the RADIUS server to a specific *pam.d* entry. FreeRADIUS defaults this string to RADIUS:

```
DEFAULT Auth-Type := Pam, Pam-Auth == "hasselltech-radius", NAS-IP-Address == 127.0.0.1
    Service-Type = Framed-User,
    Framed-Protocol = PPP,
    Framed-IP-Address = 255.255.255.254,
    Filter-Id = "15intonly",
    Framed-MTU = 1500,
    Framed-Compression = Van-Jacobson-TCP-IP
```

Ensure that your compiler's settings are configured to enable PAM support when you first begin your FreeRADIUS installation.

Open your *radiusd.conf* file and scroll to the modules section. Enable PAM functionality by examining the *pam* section inside the modules divider. The value for the *pam_auth* string corresponds with a file in the */etc/pam.d* directory on your system. Enter a name here, and make a note of it, as shown in this example:

```
pam {
                #
                #  The name to use for PAM authentication.
                #  PAM looks in /etc/pam.d/${pam_auth_name}
                #  for its configuration.  See 'redhat/radiusd-pam'
                #  for a sample PAM configuration file.
                #
                #  Note that any Pam-Auth attribute set in the 'users'
                #  file overrides this one.
                #
                pam_auth = radiusd
}
```

In the same file, scroll down to the authentication section and make sure the *pam* line is not commented out:

```
authenticate {
       pam
        unix
#       ldap
#       mschap
#       eap
}
```

Now, navigate to the */etc/pam.d* directory on your system and create a file with the same name you specified in the

# 6.2 Proxying and Realms

FreeRADIUS can act as a proxy server that adheres to the RFC specifications. To use realms, a user will likely dial in with a preferred syntax as discussed in Chapter 2: commonly, this is in the format of *user@realm* or *realm/user*. To configure the proper syntax for your implementation, consult the realm module configuration section of the *radiusd.conf* file (in the */etc/raddb* directory).

Further realm configuration takes place in the */etc/raddb/proxy.conf* file. There is also another file, */etc/raddb/realms*, but the developers of FreeRADIUS suggest using the more expandable and functional *proxy.conf* file for this purpose. The *proxy.conf* file lists various settings and configuration directives for the proxy functionality, as well as a realm configuration section in which you detail which realms belong to which authentication hosts. For example, for the realm *ralint*, the following entry would be added to the *proxy.conf* file:

```
realm ralint {
    type        = radius
    authhost    = radius.raleighinternet.com:1645
    accthost    = radius.raleighinternet.com:1646
    secret      = triangle
    nostrip
}
```

You can also configure local realms whose authentication requests are not proxied. In this case, you don't need to list a secret in the configuration. For instance:

```
realm durhamnet {
    type= radius
    authhost= LOCAL
    accthost= LOCAL
}
```

A NULL realm can be used for authentication requests without a realm specified. A NULL entry might look something like this:

```
realm NULL {
    type= radius
    authhost= radius.raleighinternet.com:1645
    accthost= radius.raleighinternet.com:1646
    secret= triangle
}
```

Finally, much like in the *users* file, there can be a DEFAULT entry that will apply to all other realms not explicitly matched. Here is an example:

```
realm DEFAULT {
    type= radius
    authhost= radlocal.corp.raleighinternet.com:1645
    accthost= radlocal.corp.raleighinternet.com:1646
    secret= iamnotamicrosoftmachine
}
```

There exist several more options with which you can configure proxying and realm functionality in the *proxy.conf* file. Table 6-1 lists the options.

Table 6-1. Realm and proxy configuration options

| Option | Description |
|---|---|
| | |

# 6.3 Using the clients.conf File

In Chapter 5, I configured a very basic FreeRADIUS system using the plain-vanilla *clients* file. That file is obsolesced by the more flexible *clients.conf* file. It's very simple to configure, however.

There are two types of entries in the *clients.conf* file: clients and NASes, or more generally, RADIUS client equipment. Clients are standard requestors used in most authentication scenarios. In the case of a client entry, the canonical name or IP address of the original source request will be matched to an entry in the *clients.conf* file, and the secret will be compared to verify the integrity of the request. A NAS entry is used for all RADIUS client equipment where it's actually a NAS or another type of client. The NAS entry changes the criteria by which request information is compared to an entry: NAS entries use the NAS-IP-Address attribute in the original source request to match the appropriate entry and then progress to the NAS-Ident attribute.

A sample complete *clients.conf* entry shown here:

```
client 172.16.1.55 {
    secret      = donttellanyone
    shortname   = totalcontrol
    vendor      = 3comusr
    type        = tc
    login       = !root
    password    = changeme
nas 172.16.1.66 {
    secret      = iamanas
    shortname   = max6000
    vendor      = lucent
    type        = ascend
    login       = !root
    password    = changeme
```

# 6.4 FreeRADIUS with Some NAS Gear

For a variety of reasons, vendors have been known to not adhere to RFC specifications. Often their products are based on an early draft of a proposed specification, sometimes vendors fail to update their products to the revised guidelines, and sometimes vendors simply choose to ignore the specification entirely. In any case, as an administrator you must cope. Unfortunately, the concept of vendor-specific irregularities and peculiarities is not foreign to NAS gear.

This section is designed to at least familiarize you with the vagaries of using some models of terminal server equipment with FreeRADIUS. Wherever possible, I will offer a workaround, another option, or some other recommendation to assist you in compensating for the problem.

## 6.4.1 Ascend Equipment

Traditionally, the attributes specific to Ascend terminal server gear are sent by FreeRADIUS as vendor-specific attributes, as per the RADIUS RFC. However, the Ascend NAS equipment itself sends its own attributes (those that are specific to the Ascend equipment) as regular, global space attributes, which, of course, causes problems with other attributes as specified in the RFC. If you suffer from a problem related to Ascend's non-standard way of dealing with its specific attributes, you will see invalid Message-Authenticator messages in your log files.

There are two options to fix this problem. The first is to enable support for vendor-specific attributes on the Ascend equipment. There are different steps to follow depending on which model of terminal server you have. If your model is the Max6000 or Max4000 series with the menu-style TAOS interface, follow these instructions:

1. Go to Ethernet, select *Mod Config*, and then choose *Auth*.
2. Find the *Auth-Compat* option at the bottom of the menu. Change this from its current setting, *OLD*, to *VSA*.
3. Save the change to make it active.

If you have the Max TNT model or the Apex 8000 series with the command-line-driven TAOS system, execute the following commands from a shell prompt.

```
 nas> read external-auth
nas> set rad-auth-client auth-radius-compat = vendor-specific
nas> set rad-acct-client acct-radius-compat = vendor-specific
nas> write
```

The other option is to perform the opposite change: enable the *old* attributes on the FreeRADIUS machine. This is a bit easier to do, since all that is required is preceding the Ascend attributes with *X-* wherever they're found. For example, the vendor-specific attribute Ascend-Data-Filter would become, in old-style attribute naming, X-Ascend-Data-Filter. It's worth noting that some Cisco equipment has the capability to emulate Ascend NAS gear with 100% compatibility, so consider whether you have mixed gear when choosing the option to rid yourself of the Ascend integration problems.

# 6.5 Using MySQL with FreeRADIUS

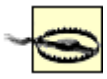
Many FreeRADIUS users have been toying with interacting *radiusd* with MySQL, which is a wonderful open source database product. Using a database allows the administrator to query data and produce reports after transactions are complete using a standard language, SQL, which is supported across platforms. Also, a database allows users and passwords to be kept in a central place, and other services can access it and make said database an extensible, complete resource. Additionally, it's a centralized administration point, which reduces the administrative headache of offering a service to the public. This section describes one possible setup to allow FreeRADIUS to authenticate against a user database held inside MySQL.

By using MySQL, you put the contents of the *users* file inside the database, and instead of storing all of the user information in one file, with separate stanzas for each user, the data will now exist in several different database tables. This majorly improves speed and scalability and offers a modicum of flexibility, too.

First, download, compile, and install MySQL for your RADIUS machine. There are several web resources available to assist you in doing this:

- The MySQL web site (http://www.mysql.com) offers database downloads as well as API information, graphical tools to manage the database, applications contributed by third parties, and complete documentation for the core database product.

- There is also a convenient Windows-based tool to manage a remote MySQL database called SQLion ( http://www.exxatools.com/SQLion.html) that will make it easy to create and populate tables. Of course, in lieu of a desktop-based product, there is also the venerable Linux tool, *phpMyAdmin* ( http://www.phpwizard.net/projects/phpMyAdmin/), which can be used over the Web for much the same purpose.



It is imperative that you have the *mysql-devel* package installed (with headers and libraries included) before compiling and installing FreeRADIUS. If you don't, *radiusd* will not compile with MySQL support properly.

To begin the rest, follow these steps:

1. Download, compile, and install FreeRADIUS. This process is detailed in Chapter 5. Using MySQL in conjunction with *radiusd* doesn't call for any special compile-time or install-time flags, so a vanilla installation should function correctly.

2. Configure the test RADIUS system, also as described in Chapter 5. You will want to add a user in the shell (use the useradd command) to use for authentication purposes. The remainder of this section will assume you created a user "radius" in the system that belongs to a group "radius."

# 6.6 Simultaneous Use

Recall from Chapter 1 that RADIUS is a stateless protocol. Additionally, because of the way RADIUS accounting works, it's entirely possible and even probable that a RADIUS server will have an internal list of who is currently logged on that is different than the actual state of the RADIUS client ports—in other words, your RADIUS server may think users are logged on when they really aren't, and vice versa. Fortunately, most NAS equipment includes some mechanism by which the administrator (or the RADIUS daemon servicing authentication requests) can query it to find out which user is assigned to what port. This could be done through Telnet, the deprecated finger protocol, or even the Simple Network Monitoring Protocol (SNMP).

This ability is especially important when attempting to control multiple logins at the same time from the same user. There exists a utility to tell FreeRADIUS to check on the terminal server first to see if a user is already logged on before denying his request to log on, thereby compensating for the RADIUS accounting discrepancies. The best way to do this is by installing two modules—the *SNMP_Session* and BER modules—from the popular traffic-monitoring program MRTG. (These are core Perl modules, actually.) Having those modules installed lets a utility included in FreeRADIUS, the *checkrad* script, communicate with the terminal server equipment directly using the SNMP protocol. You can obtain more information and download these modules from the "SNMP Support for Perl 5" web site at http://www.switch.ch/misc/leinen/snmp/perl/.

If you have USR/3Com Total Control terminal server gear and you want to make use of the checking routine, you will need the Net::Telnet module for Perl 5. This can be obtained from the CPAN archive at http://www.perl.com/CPAN/.

To enforce a simultaneous-use restriction, you need to add a parameter to either an individual user's entry or a DEFAULT entry in the RADIUS *users* file (*/etc/raddb/users*). The value of the Simultaneous-Use attribute is the number of sessions that can occur at the same time with the same username. To enforce a restriction on user *awatson*, for example, of two simultaneous connections, I would configure a user entry for her similar to the following:

```
Awatson     Auth-Type := System, Simultaneous-Use := 2
            Service-Type = Framed User
            <continue attribute listing>
```

You can also define a certain group of users—for example, a multilink group that can have two logins concurrently—while the rest of the user base can only have one simultaneous session. To achieve this, use the following DEFAULT entries and the fall-through feature:

```
   DEFAULT      Group == "multilink", Simultaneous-Use := 2
                Fall-Through = 1
  DEFAULT       Simultaneous-Use = 1
                Fall-Through = 1
```

Once this is configured, the server now knows to use the *checkrad* script (located at either */usr/local/sbin/checkrad* or */usr/sbin/checkrad*). When does it invoke the script? When a user connects, FreeRADIUS looks in its list of currently active users, which is kept in */var/log/radutmp*. (Executing *radwho* at a command prompt will display the contents of this file on the screen.) If it finds that the username associated with the pending request is already listed in *radutmp*, then it will execute the *checkrad* script. The *checkrad* script then communicates with the NAS gear via finger, Telnet, or SNMP and determines whether that user is indeed logged on. It then either accepts or denies the request for a concurrent session based on the value of the Simultaneous-Use attribute as configured in the *users* file.

Be forewarned that the load and performance impact of using *checkrad* can be quite

# 6.7 Monitoring FreeRADIUS

Part of proactive system administration is monitoring for problems before they occur. While you, as the administrator, are probably at your office and within reach of the RADIUS server for 8-12 hours a day, the remaining hours aren't devoid of users who depend on your service. What happens when (not if) your FreeRADIUS server has a problem and you're not around?

This section describes using some freely available tools to set up FreeRADIUS such that if it happens to shut down because of an error, it automatically restarts. While it's still your responsibility to troubleshoot the problem, it does recover the service so you don't have to deal with angry users calling because they can't get on the Internet.

Let's use Dan Bernstein's DaemonTools package, and in particular, its "supervise" service to monitor *radiusd*. To get started, surf to the DaemonTools web site at http://cr.yp.to/daemontools.html, download the package, and install it. Dan has complete installation instructions on his site as well as a good deal more documentation that outlines and details the capabilities of DaemonTools. That's beyond the scope of this application, but it's likely you can find a use for some of the service management that DaemonTools provides.

Once the tools are installed, you need to create a RADIUS service directory that DaemonTools can use. It's common practice to place this directory on the */var* partition in the *svc* directory, although it can be placed anywhere you choose. For the rest of this section, I'll assume you chose the */var/svc/radiusd* directory. Make the directories, and then open up your favorite text editor.

In the text editor, you need to create a simple shell script, called *run*, that will call up *radiusd* in the event it fails. Here's a sample:

```
#!/bin/sh
exec /usr/local/sbin/radiusd -s -f
```

Of course, replace these directories with ones appropriate for your machine as needed. The -f flag is important in this case; it tells FreeRADIUS to stay on the console screen and not return to a command prompt. If it detaches itself, DaemonTools will think it died and attempt to restart it using the shell script provided above.

Now, make that script executable:

```
chmod +x /var/svc/radiusd/run
```

Finally, tell DaemonTools to watch FreeRADIUS.

```
Supervise /var/svc/radiusd.
```

DaemonTools is now setup and will restart *radiusd* upon its death.

Table 6-7 lists additional maintenance commands, available from the DaemonTools *svc* utility, that you will likely find useful.

Table 6-7. FreeRADIUS service management commands

| **Action** | **Command** |
|---|---|

# Chapter 7. Other RADIUS Applications

The previous two chapters have focused on using the FreeRADIUS product as the basis of an authentication/authorization/accounting system for a regular Internet service provider-style setup. In this chapter, I'll cover FreeRADIUS in conjunction with Web, LDAP, and email servers, and will discuss a utility, RadiusReport, for parsing RADIUS accounting files to glean valuable information from them.

# 7.1 RADIUS for Web Authentication

Chances are good that you have an area of your web site that needs to be protected from general public access. If you use the Apache web server, you may be familiar with the various methods by which this can be done: using an *.htaccess* and *.htpasswd* combination, setting Unix file system permissions, using Allow and Deny directives inside the Apache configuration file, and others. However, it's now possible to instruct Apache to authenticate against an existing RADIUS database of users, thereby protecting the area of your web site from unknown users and allowing access to those you trust.

This authentication is done using a module developed for Apache 1.x called *mod_auth_radius*. (Apache 2.0 had not been released at the time, and the module has yet to be updated for Version 2.0.) In effect, Apache becomes a RADIUS client—occupying the traditional position of the NAS in the authentication chain—and hits off the RADIUS server for authentication and accounting requests. Not only does this save administrative time by consolidating what potentially could become two user databases into one, but it also allows for more flexibility. Namely, RADIUS accounting can be used to track usage statistics for this protected site. Apache can keep detailed logs, but sometimes it's helpful to have all audit information in one place.

There are several potential applications for this module. The following scenarios are likely candidates for this module:

- A corporation who wants to create a special Intranet site specifically for its remote, mobile, and home users
- An Internet service provider who wishes to create a private site for subscribers only; perhaps a billing or support site that contains technical information suitable only for paying customers
- A web-based business that sells subscriptions to an online database or an online journal

And there are many others.

## 7.1.1 The Functionality

The *mod_radius_auth* module follows a predictable pattern in its use. A typical transaction occurs like this:

1. The browser submits a page request for http://www.website.com/index.html.
2. Apache sees that the directory is secured and sends an *Authorization Required* prompt (with spaces for the username and password) to the end user.
3. The user responds to the authentication request with his credentials. The browser sends the response, and the

# 7.2 Using the LDAP Directory Service

The ever-present complaint of systems administrators who deal with multiple user databases across multiple platforms is that of efficiency. Why can't all of my users be listed, configured, and managed from one set of tools? Why can't my various application servers—secured Web, email, newsgroups, and others—all tie into that one database and use its list? Without a centralized repository for user information, the effort of simply changing a password is multiplied by the number of systems on which a unique copy of the password is stored.

Fortunately, there is an answer, and better yet, it's standards based. The Lightweight Directory Access Protocol, or LDAP, is a directory-based database of information about users of a particular network. LDAP is a protocol that uses standard queries, much like SQL, to talk with a compliant backend. Using LDAP allows applications that support it to communicate with a centralized database and use its information in their internal operations. While a discussion about LDAP could fill volumes (and, in fact, has), the important fact to take away from this commentary is that FreeRADIUS has full and complete support for LDAP. This is part one of the equation. I have an LDAP client, but it needs something to talk to.

Enter CommuniGate Pro, an excellent email server product from the fine folks at Europe-based Stalker Software ( http://www.stalker.com). CommuniGate Pro is designed to run on any number of processor architectures: from the Intel x86 regime to IBM's midrange servers and OS/400 computers. The product excels in every respect: it's intelligently designed, easy to install and use, and an excellent performer. The product has been subjected to numerous benchmarks in competition with other Internet mail servers and won each test hands down. It also is a fine LDAP server and can be configured to allow other applications to query its user database in full LDAP compliance. That's part two of our equation.

How does all of this fit together? Most organizations need email functionality. Of course, you're reading this book likely because your organization provides dial-up access to end users, either for profit or as part of your regular corporate business activities. Allowing FreeRADIUS, a robust RADIUS server, and CommuniGate Pro, an excellent mail server, to communicate with each other brings you the best of both worlds: stable server platforms and interoperability to ease the headaches of administration.

In this section, I'll detail how to make FreeRADIUS authenticate against the CommuniGate Pro LDAP user database. Most of the instructions in this section can be applied to any other LDAP database product, but there are a few instructions specific to CommuniGate Pro that are detailed. You can realize the benefits of this integration with any LDAP backend, but using CommuniGate Pro gives you a powerful email server to boot. On that note, let's begin!

## 7.2.1 Configuring FreeRADIUS to Use LDAP

To instruct FreeRADIUS to use the LDAP protocol instead of PAM or another local user authentication database, you need to install the OpenLDAP product. As of this writing, the latest version of OpenLDAP is 2.0.23. To install OpenLDAP on your system, perform the following steps:

1. Download the product, preferably in *.tar.gz* form, from the OpenLDAP web site at http://www.openldap.org/software/download/.

2. Decompress the program with the following command:

# 7.3 Parsing RADIUS Accounting Files

One of the most useful aspects of RADIUS is the utility of its accounting portion. Logs from the RADIUS accounting server can be used for a multitude of purposes, including billing, usage planning, attack forensics, and auditing. Most Internet service providers have billing systems that directly import, analyze, interpret, and report the data contained within the accounting logs. But for corporate situations in which billing isn't required or for ISPs wanting information not provided by the billing system, it's useful to have a utility that will read the logs and report basic information for the outside of your standard reporting system.

Paul Gregg has created an excellent utility, written in Perl, called RadiusReport that offers this functionality. *RadiusReport* allows you to import log files and create different reports based on their contents. The utility supports the log files that FreeRADIUS generates, and it also has support for the following RADIUS servers:

- Livingston Radius, Versions 1.16, 2.0, and 2.01
- Dale Reed's RadiusNT
- Merit Radius
- Ascend Radius
- Radiator
- Novell's BorderManager Authentication Services (requires a separate utility to "massage" the format of the logs)

RadiusReport will generate all sorts of useful reports, including the projected telephone bill, reporting filtering based on specific months if you have multiple periods aggregated into a single file and parsing based on interim months. The reports are configured and constructed from command-line flags issued with the program call. The program will even read a compressed file, in case you use *gzip* or *tar* to compress and archive your old accounting logs.

RadiusReport is a Perl program, so it requires Version 5 of the language to be installed on the system. It also requires the POSIX module, which comes bundled with the Perl language in most cases. The utility needs POSIX compliance to correctly translate record date information into a timestamp field if your server doesn't make a timestamp.

RadiusReport can be downloaded from Paul Gregg's web site at http://www.pgregg.com/projects/radiusreport/.

## 7.3.1 Generating Reports

# Chapter 8. The Security of RADIUS

It's a little ironic that I'm devoting a chapter (albeit shorter than the others) to the security shortcomings of the RADIUS protocol, but it's something that needs doing. Unfortunately, RADIUS—a protocol designed from the outset to provide security so that only authorized users can take advantage of resources offered to a large group of people—has security problems, and some are actually quite serious.

The most prominent security vulnerability is rooted in RADIUS's wide use. It enjoys support from a number of network equipment vendors and is found in nearly all Internet service providers and corporate dial-up implementations. This popularity, however, is a double-edged sword. Security vulnerabilities in the core RADIUS protocol leave thousands upon thousands of systems open to compromise. Further, major changes can't be made to the core protocol, because that would run the risk of breaking compatibility with those same thousands upon thousands of systems that run RADIUS.

In this chapter, I'll discuss these vulnerabilities, offer some workarounds that protect your systems better, and close with a commentary from a security analyst on why users of RADIUS should push for minor protocol changes.

# 8.1 Vulnerabilities

It has been discovered by many that RADIUS has some fundamental flaws that may allow an attacker to compromise the integrity of a transaction. Primarily, the User-Password protection mechanism is inherently quite insecure, employing encryption and cryptographic techniques improperly. The concept of a response authenticator inside the RADIUS packet is genuinely good, but the implementation of such in the protocol is poorly designed. The Access-Request packet is not authenticated—at least as per the protocol specification—by any machine party to the transaction. The randomness of a client's generation of request authenticators is not really random enough. And finally, the shared secret is a primitive method of securing RADIUS client-to-server transactions.

Now I'll look at each of these vulnerabilities in greater detail.

## 8.1.1 MD5 and the Shared Secret

The shared secret is vulnerable because of the weak MD5 hash that hides the response authenticator. A hacker could easily attack the shared secret by sniffing a valid Access-Request packet and its corresponding response. He can easily get the shared secret by pre-computing the MD5 calculation from the code, ID, length, request authenticator, and attributes portion of the packets and then resuming the hash for each guess he makes.

## 8.1.2 The Access-Request Packet

There is no verification or authentication of the RADIUS Access-Request packet, as per the RFC specification, by default. The RADIUS server will perform a check to ensure that the message originated from an IP address listed as one of its clients, but in this day and age, spoofed IP addresses are easy to find and use. This is a serious limitation of the RADIUS protocol design.

As of now, the only workable solution is to require the presence of the Message-Authenticator attribute in all Access-Request messages. Briefly, the Message-Authenticator is the MD5 hash of the entire Access-Request message, using the client's shared secret as the key. When a RADIUS server is configured to only accept Access-Request messages with a valid Message-Authenticator attribute present, it must silently discard those packets with invalid or missing attributes. More information on the Message-Authenticator attribute can be found in Chapter 9 or in the RFC 2869.

If your implementation somehow prevents the use of the Message-Authenticator attribute, at least consider using some sort of account-lockout feature, which disables authentications after a specified number of authentication attempts within a specified time.

## 8.1.3 The User-Password Cipher Scheme

The way in which the User-Password attribute is handled, on a very general basis, is known as a stream cipher. A *stream cipher* is an encryption method that works with continuous streams of input, which is usually a stream of plain-text bits rather than fixed blocks; its opposite is a *block cipher* , which is an encryption method that processes input in fixed blocks of input, which are typically 64- or 128-bits long. A stream cipher generates a *keystream*, and this is used in the encryption: when you combine this *keystream* with the plain-text input stream using the XOR operation, the contents of the stream are encrypted. The generation of the *keystream* can be independent of the plain text and *ciphertext*, yielding what is termed a synchronous stream cipher, or it can depend on the data and its encryption, in which case the stream cipher is said to be self-synchronizing.

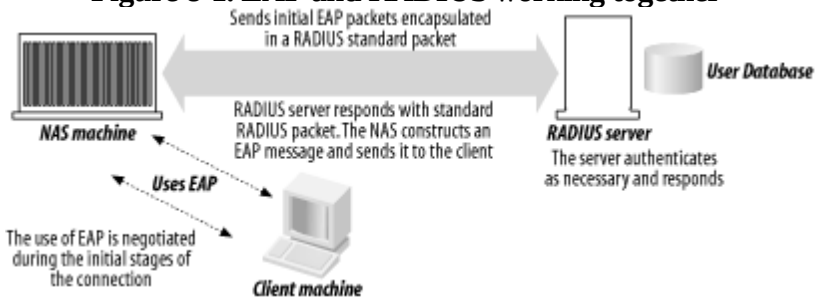
# 8.2 The Extensible Authentication Protocol

EAP is an extension to the PPP protocol that enables a variety of authentication protocols to be used. EAP is not tightly bound to the security method. It passes through the exchange of authentication messages, allowing the authentication software stored in a server to interact with its counterpart in the client. EAP serves as a sort of replacement protocol, allowing the initial negotiation of an authentication protocol (such as CHAP and MS-CHAP Versions 1 and 2) and then the agreement on both ends of the connection on a link type, which is a specific EAP-authentication scheme. Once these two elements have been confirmed, EAP allows for an open-ended conversation between a RADIUS server and its client.

EAP is designed to function as an authentication "plug-in," with libraries on both the client and the server end of a PPP connection. Each authentication scheme is associated with a particular library file, and once a specific library has been dropped into place on both ends, that new scheme can be used. Thus, the protocol can easily be functionally extended by vendors at any time without having to redesign the whole protocol. EAP currently supports authentication schemes such as Generic Token Card, OTP, MD5-Challenge, and Transport Level Security (TLS) for use in smart-card applications and support for certificates. In addition to supporting PPP, EAP is also supported in the link layer as specified in IEEE 802. IEEE 802.1x defines EAP's use in authenticating 802 devices, like WiFi access points and Ethernet switches.

How does EAP relate to RADIUS? EAP secures RADIUS more. Using RADIUS with EAP is not an official authentication scheme of EAP; rather, look at it as the passing of EAP messages of any EAP type by the RADIUS client gear and the RADIUS server. EAP over RADIUS is typically set up in this fashion: the access server is configured to use EAP and also to use RADIUS as its authentication provider. When a service consumer attempts to connect, the service consumer negotiates the use of EAP with the RADIUS client gear. The end user then sends an EAP message to the RADIUS client, and the RADIUS client encapsulates the EAP message as a RADIUS message and sends it to the RADIUS server. The RADIUS server acts on the encapsulated message and sends a RADIUS-style message back to the RADIUS client. The RADIUS client then constructs an EAP message from the RADIUS message and sends it back to the service consumer/end user. Figure 8-1 illustrates this flow.

**Figure 8-1. EAP and RADIUS working together**



# 8.3 Compensating for the Deficiencies

All of the security issues presented in this chapter have workarounds. Some have been listed within the discussion of each vulnerability, but this section serves as a quick reference checklist, from which you can decide which workarounds to employ in your RADIUS implementation. This section outlines some of the basic steps you can use to compensate for some of the more nefarious RADIUS design decisions:

*Use the IPsec protocol with ESP and an encryption algorithm such as 3DES.*

When IPsec encrypts the whole RADIUS message, fields open to compromise—namely the request authenticator fields and the User-Password, Tunnel-Password, and MPPE-Key attributes—cannot be viewed. To decrypt these fields, an attacker first must break into the ESP-protected message. This protects the entire RADIUS message and keeps it from prying eyes.

*Require any shared secrets in use to be either 22 keyboard characters long or 32 hexadecimal digits long.*

This protects against the deficiencies and the unprotected nature of the shared secret concept.

*Use a different shared secret for each RADIUS client and server pair.*

This is just a basic security measure, much like having a different password for a variety of web sites and computing resources.

Use the Message-Authenticator attribute in all Access-Request messages. On the client side, make sure the Message-Authenticator is used and ensure it can be configured.

On the server side, require that the Message-Authenticator attribute be present and also allow here for its configuration. This compensates for having no Access-Request messages authenticated anywhere along the transaction path.

*Use a cryptographic-quality random number generator to generate the request authenticator.*

This offsets the rather limited quality of the request authenticator's implementation.

You might also consider protecting the links from the end user to the RADIUS client gear using EAP and one of the strong encryption types available with its use. For example, you could use EAP-TLS, which is a strong EAP method that requires the exchange of client and RADIUS server certificates. The use of EAP messages inherently requires a valid Message-Authenticator certificate, which protects messages that can't otherwise be protected by the use of IPsec.

Also, along with EAP, think about using mutual authentication methods. Very simply, both ends of the connection authenticate their peer in mutual authentication. The authentication attempt is rejected if either end's authentication fails. EAP-TLS is a mutual authentication method: the RADIUS server validates the user certificate of the client, and the client validates the computer certificate of the RADIUS server.

Finally, if the PAP authentication protocol is not required, disable it on both the client and the server end. PAP should only be used as a secure connection when it's used in conjunction with OTP and Token Card authentication where the password is reasonably complex and changes with each use. However, even in this situation, having PAP enabled allows for misconfigured end users to negotiate with the RADIUS client gear and at that point, they could potentially send unprotected passwords. If at all possible, use EAP with the OTP and Token Card authentication types instead of PAP. In the same line of thinking, disable LAN Manager encoding if you use MS-CHAP.

# 8.4 Modifying the RADIUS Protocol

It may be frustrating to have to employ workarounds to inherent deficiencies in the RADIUS protocol. As informed, knowledgeable RADIUS users (and you are knowledgeable now that you are reading this book), we need to push for a protocol revision. Joshua Hill, of InfoGard Laboratories, eloquently makes a case for a revision in the following mini-essay.

So, why attempt to modify RADIUS at all? Why not just go to another (presumably more modern and more secure) protocol? Well, for the most part, the answer is, "because such a protocol doesn't currently exist." In the near future, however, Diameter is likely to be released by the IETF.

Diameter is the planned RADIUS replacement. The great majority of all the protocol work that has gone into Diameter has been directed at removing some of the functional limitations imposed by the RADIUS protocol. Effectively, no work has been done that relates to the client/server security of the protocol. (CMS is defined, but this is a security layer for the proxy to proxy interaction, not the client to proxy/server interaction.)

So, does this mean that they continue to use even RADIUS' ad hoc system? No: they removed all security functionality from the protocol. In essence, the developers did the protocol designer's equivalent of punting. Section 2.2 of the current Diameter protocol spec says:

"Diameter clients, such as Network Access Servers (NASes) and Foreign Agents MUST support IP Security, and MAY support TLS. Diameter servers MUST support TLS, but the administrator MAY opt to configure IPSec instead of using TLS. Operating the Diameter protocol without any security mechanism is not recommended."

So, IPSec and/or TLS handle all security aspects of the protocol. From a security aspect, this strikes me as a very good idea. Both IPSec and TLS are fully featured (sometimes too fully featured) protocols that many people have reviewed. That's already much better than RADIUS ever did.

Examining this from a slightly different angle gives me some cause for concern, however. It strikes me that the overhead imposed by a full TLS/IPSec implementation is very significant for many current-day embedded devices. This would seem to indicate that (at least in the near future) manufactures are going to either continue to use RADIUS or ignore the Diameter standard and perform Diameter without TLS or IPSec.

Because of this, I suspect that it would be advantageous to push for at least minimal RADIUS protocol revision.

# Chapter 9. New RADIUS Developments

Up to this point, I've covered the contents and specifications of the original RADIUS RFC drafts. Since those drafts were approved and published, new advancements in technology have mandated some revisions to those RFCs, particularly in the areas of tunnel support and new security technologies. In this chapter, I'll cover these updates and how they might affect your current implementation or any changes you will make in the future.

# 9.1 Interim Accounting Updates

RADIUS now includes support for interim accounting updates. Prior to the issuing of the RADIUS Extensions RFC in June 2000, accounting updates were done primarily at the beginning and end of a transaction, when the server received Accounting-Start and Accounting-Stop packets from the user. However, now the server can include the Acct-Interim-Interval attribute in the message. The value of this attribute is the time (in seconds) between accounting update messages. An administrator can also choose to configure a minimum value locally on the RADIUS client, but this value always overrides any Acct-Interim-Interval value found in an Access-Accept packet.

This attribute can include all the attributes found in the standard Accounting Stop message except the Acct-Term-Cause attribute. The data sent within the Acct-Interim-Interval packet is always cumulative; that is to say, the data in each interim update contains data from the start of the session through the current state of the session at the time the packet is sent. Because this data is cumulative, it's up to the RADIUS client gear to ensure that only one interim update packet exists on the wire at once. Some RADIUS client machines may choose to add a delay of some amount of seconds to make sure that the previous condition is satisfied.

# 9.2 The Apple Remote Access Protocol

The Apple Remote Access Protocol (ARAP) sends traffic based on the AppleTalk protocol across PPP links and ISDN switched-circuit networks. ARAP is still pervasive in the Apple market, although the company is attempting to transition into an Apple-specific TCP stack for use over a PPP link. ARAP support is typically found in most RADIUS client gear, and RADIUS now supports authenticating based on the ARAP protocol.

ARAP authentication typically takes one to two steps, as follows:

1. The first step is basically a mutual authentication with an exchange of random numbers signed with a *key*, which happens to be the user's password. The RADIUS client challenges and authenticates the dial-in client, and the dial-in client challenges and authenticates the RADIUS client challenges. First, the RADIUS client sends random numbers of 32 bits to the dial-in client inside an ARAP msg_auth_challenge packet. Then, the dial-in client uses his password to encrypt the two random numbers sent by the RADIUS client with DES. The dial-in client sends the result back in a msg_auth_request packet. Finally, the RADIUS client unencrypts the message based on the password it has on record for the user and verifies the random numbers are intact. If so, it encrypts the challenge from the dial-in client and sends it back in a msg_auth_response packet.

2. The RADIUS client may initiate a second phase of authentication using optional *add-in security modules*, which are small pieces of code that are run on both ends of the connection and provide read and write access across the link. Some security token vendors use these add-ins to perform their own proprietary authentication.

There are some caveats to integrating ARAP and RADIUS based on the way ARAP is designed. Namely, ARAP transmits more security profile information after the first phase completes but before the second phase of authentication begins. The profile information is contained within a single attribute and is a series of numeric characters relating to passwords. Even so, challenge responses and this new profile information must exist at times that may seem a bit non-standard. But it is the standard.

To allow an ARAP-based client access to the resources the RADIUS server is protecting, an Access-Request packet must be issued on behalf of the ARAP client. This process takes place as one would imagine: the RADIUS client with the ARAP protocol generates a challenge based on a random number and, in response from the end-user client, receives the challenge and the username. The relevant data is then forwarded to the RADIUS server inside a standard RADIUS Access-Request packet. The data that is transplanted is as follows: User-Name, Framed-Protocol with a value of 3 for ARAP, ARAP-Password, and any other pertinent information like Service-Type, NAS-IP-Address, NAS-Id, NAS-Port-Type, NAS-Port, NAS-Port-Id, Connect-Info, and others. Note that only one of the User-Password, CHAP-Password, or ARAP-Password attributes needs to be present in the Access-Request packet. Any EAP-Messages attributes supercede any of those attributes' presence in a packet.

The authentication then takes place. If the RADIUS server doesn't support ARAP, it should return an Access-Reject message. If it does, then in order to authenticate the user it should verify the user response using the challenge, found in the first eight octets of the request authenticator, and the associated response, found in the first eight octets of the ARAP-Password attribute. The resulting Access-Accept message, if this information is verified and the user is indeed successfully authenticated, should be formatted in the following manner:

-

# 9.3 The Extensible Authentication Protocol

EAP is supported in the new RADIUS extensions and allows for new authentication types to be used over links running on the PPP protocol. Authentication schemes such as public key, smart cards, one-time passwords, Kerberos, and others are supported over PPP when EAP is used. To support EAP, RADIUS includes two new attributes—EAP-Message and Message-Authenticator—that are described in this section.

Typically, the RADIUS server acts as an intermediary between the client and a backroom proprietary security and authentication server. It normally encapsulates the EAP packets within a standard RADIUS packet, using the EAP-Message attribute, and then transmits them back and forth between the two machines. This lets the RADIUS server talk to the other proprietary authentication server using a standard protocol that requires no special modifications on the RADIUS server. It can still fully support standard RADIUS requests with reduced overhead.

A typical EAP over RADIUS transaction occurs in a standard format, which is outlined here:

1. The dial-up client and the RADIUS client gear negotiate the use of EAP within their specific link control protocol—this is most commonly PPP.
2. The RADIUS client then sends an EAP-Request/Identity message to the client unless its identity has been verified through some other means, such as callback or caller ID.
3. The dial-up client then responds with an EAP-Response/Identity message.
4. The RADIUS client gear receives this response from the client and forwards it to the RADIUS server inside a standard RADIUS Access-Request using the EAP-Message attribute.
5. The RADIUS server responds with a standard Access-Challenge packet that contains an EAP-Message attribute. The EAP-Message attribute contains a full EAP packet.
6. The RADIUS client gear unwraps the encapsulated EAP message and forwards it to the dial-up client.

The authentication continues as many times as needed until either an Access-Reject message or an Access-Accept message is received. If the EAP transaction follows these typical steps, then the RADIUS client gear will never have to manipulate an EAP packet. Of course, the world is not always as simple as that; as such, there are a couple of caveats to this scenario.

First, you must permit proxy capability between RADIUS machines that may not be compliant with the EAP protocol. If the RADIUS client machine sends the EAP-Request/Identity, as described previously in step two, the

# 9.4 Tunneling Protocols

With the advent of work-from-home strategies and the branch-office concept becoming ever more popular, the dependence on access to corporate networks and privatized ISPs has become stronger. There exists a way to use a sort of tunnel to log in to corporate network over the Internet and access that network's resources as though you were locally attached to it. Although discussing tunnels is beyond the scope of this book, RADIUS does support a variety of tunneling protocols, both voluntary and compulsory. New RADIUS attributes were introduced with RFC 2868 that provide support for this emerging technology.

As well, private ISPs and even some corporate IT data centers want to be able to account for the use of their service for accounting, billing, and auditing purposes. RADIUS accounting, of course supporting the AAA model as discussed in Chapter 1, is an obvious way to collect this data, especially with the new tunneling-support attributes, some modifications to the Acct-Status-Type attribute, and some entirely new attributes specifically focused at RADIUS accounting.

The new values for the Acct-Status-Type attribute are listed in Table 9-1.

Table 9-1. New values per RFC 2867 for Acct-Status-Type

| Value | Name | Description | Also requires |
|---|---|---|---|
| 9 | Tunnel-Start | Marks the creation of a tunnel with another end point. | User-Name, NAS-IP-Address, Acct-Delay-Time, Event-Timestamp, Tunnel-Type, Tunnel-Medium-Type, Tunnel-Client-Endpoint, Tunnel-Server-Endpoint, Acct-Tunnel-Connection |
| 10 | Tunnel-Stop | Marks the destruction of a tunnel with another node. | User-Name, NAS-IP-Address, Acct-Delay-Time, Acct-Input-Octets, Acct-Output-Octets, Acct-Session-ID, Acct-Session-Time, Acct-Input-Packets, Acct-Output-Packets, Acct-Terminate-Cause, Acct-Multi-Session-Id, Event-Timestamp, Tunnel-Type, Tunnel-Medium-Type, Tunnel-Client-Endpoint, Tunnel-Server-Endpoint, Acct-Tunnel-Connection, |

# 9.5 New Extensions Attributes

In the familiar (yet repetitive, I know) format of Chapter 2, I will now detail the new attributes offered in RFC 2869, as well as those specified in the "RADIUS Attributes for Tunnel Protocol Support" (RFC 2868) and "RADIUS Accounting Modifications for Tunnel Protocol Support" (RFC 2867). They are presented in ascending order of the attribute number.

### *Acct-Input-Gigawords*

| | |
|---|---|
| Attribute Number | 52 |
| Length | 6 |
| Value | INTEGER |
| Allowed in | Accounting-Request |
| Prohibited in | Access-Accept, Access-Request, Access-Reject, Access-Challenge, Accounting-Response |
| Presence in Packet | Not required |
| Maximum Iterations | 1 |

The value of this attribute is the number of times that the Acct-Input-Octets counter has exceeded and wrapped over 232 since this transaction's inception. It can only be present in Accounting-Request packets where the value of the Acct-Status-Type is either *Stop* or *Interim-Update*.

### *Acct-Output-Gigawords*

| | |
|---|---|
| Attribute Number | 53 |
| Length | 6 |
| | |

# Chapter 10. Deployment Techniques

It's the do-or-die moment: it's time to deploy your AAA infrastructure. That infrastructure most likely takes the form of one or more RADIUS servers (otherwise you would probably not be reading this book). This chapter is designed to cover many of the inevitable questions that come up with regard to designing a plan to deploy RADIUS servers.

First, I'll look at configuring the typical services that are offered by ISPs and corporations to their clients and then broaden that to cover extended services that support other business models. Next, I'll discuss how to maintain the service by designing a secure, highly available network. Following that are two case studies of RADIUS implementation design. Finally, I'll provide information about other RADIUS servers, available documentation, and other resources you can use to support your RADIUS operation.

# 10.1 Typical Services

As you've learned from the chapters on FreeRADIUS, the users that connect through your RADIUS server must be either configured into the *users* file for the RADIUS server itself or known by a remote system with which the initial RADIUS server can communicate. Anything else falls into the default connection configuration, which is sometimes known as the "catchall." Most implementations have a generic configuration that is meant for most users and a few user-specific configurations sprinkled about. In the following sections, I will provide examples of both scenarios whenever appropriate.

## 10.1.1 System Shell Accounts

The shell account, a popular service 5 to 10 years ago but less so now, is a unique type of connection that allowed access to the command line of a remote server. Users would dial in to some NAS gear, which would open a channel to the remote "shell server," and it would then prompt the user for authentication information. Assuming he provided proper credentials, the user was authenticated, got a shell prompt on the remote machine, and the NAS acted as the pass through from the client to the server. That's an important distinction, since with shell accounts the user is not provided with a direct IP address for the remote network. Since he doesn't have his own IP, he must talk with a system that does in this scenario.

There are two common types of protocols used to connect to shell accounts on machines: *Rlogin* and *Telnet*. *Rlogin* was more popular, since it was the most configurable of the two, but *Telnet* is more secure. The RADIUS server, however, must be prepared to support both protocols. An example configuration stanza from the RADIUS users file for shell account access is listed in Example 10-1.

**Example 10-1. RADIUS configuration for shell accounts**

```
Jonathan
    Service-Type = Login,
    Login-Service = Telnet,
    Login-IP-Host = 172.16.1.37

Anna
    Service-Type = Login,
    Login-Service = Rlogin,
    Login-IP-Host = 172.16.1.38
```

Of course, you can default the configuration—meaning all users will use the same configuration, with *Rlogin*—by using the excerpt shown in Example 10-2.

**Example 10-2. Default shell account configuration**

```
DEFAULT
    Service-Type = Login
    Login-Service = Rlogin,
    Login-IP-Host = 172.16.1.38
```

## 10.1.2 Direct Connect Accounts

Today, you'll find most ISPs provide direct connect accounts using a framed remote access protocol such as SLIP or PPP. With these accounts, the connecting user is assigned an IP address (or, in the case of static IP addresses, allowed to use an address) on the remote network, so that she may function like an actual node on that network.

SLIP and PPP are both available for these kinds of connections, although usually PPP is used now, since it has many benefits: it is better supported, more robust, and has quite a few link negotiation features that SLIP just doesn't have.

# 10.2 RADIUS and Availability

High availability has become the latest buzzword in Internet service. Advertisements abound for network operation centers (NOCs) with triple-capacity electric generators, dual HVAC systems, geographical dispersion, waterless combustion control, and other facilities to handle problems. While these certainly are methods to obtain and retain high availability, it seems that sometimes people lose sight of the point of such exercises: to maintain the existence and offering of services when others systems on all "sides" of the service are failing. I say "sides" to refer to the hierarchical tree in which most systems reside: there are often machines relying on a specific box, and that box relies on other boxes, and it also may work in tandem with others.

There are several strategies for planning for failure, which is the main tenet in high availability. The one most disaster-planning experts use is to account for what would be a worst-case scenario for your implementation. There are several questions to ask yourself when designing a highly available system:

*Am I familiar with the normal traffic and availability of my systems?*

Am I aware of the inherent weaknesses my implementation has? You need to know what the normal behavior of your system is when deciding how best to concentrate your efforts to make it available.

*Do I have a single point of failure in my network?*

That is, is there one device that provides such critical service that if it went down, users could not obtain the service they need? Single points of failure are disastrous to all kinds of redundancy because they make it moot: if your system goes down, it's completely unavailable.

*What events could coincide that would overwhelm the capacity of my network?*

This scenario often comes into play when a downed system that is not part of the implementation causes certain events to happen inside the system. You'll see more of this later in the chapter.

*How can I eliminate single points of failure?*

Would several systems performing the same task as the same time (a cluster) cure this ailment? Conversely, what systems can fail without bringing down the entire network? Prioritizing the systems to which you apply availability strategies helps you keep the cost in check while ensuring the greatest possible uptime for your system.

*How can I be proactive about reducing errors and outages?*

It should be no surprise to an administrator that most errors considered catastrophic to a network are the cause of events that have been long in the making. Monitoring your systems for potential errors and their indications help to ensure problems are handled and eliminated before they even become problems.

These questions give you a fairly complete estimate of your implementation's weak points, both inside and outside of your control. Let's step through each of the questions with regard to designing a RADIUS implementation.

## 10.2.1 Determining Normal System Behavior

To establish a proper and accurate baseline for your system, there are two types of requirements you need to consider: explicit requirements, which are those mandated by your users or your company's management; and derived (or implicit) requirements, which mainly stem from the explicit requirements. For example, you may be required to make all reasonable efforts to have service restored within 15 minutes of downtime. The 15-minute window is an explicit requirement. However, you may also require that your systems have hot-swap hard drives so that you can indeed switch out a dead disc within 15 minutes. Your hot-swap requirement is derived from the explicit requirements.

# 10.3 Other Things RADIUS

There are various other "mini-facets" of RADIUS that I haven't touched on in this book. This section is designed to point you to alternative RADIUS servers, special RADIUS tools that are available to help you with your deployment and day-to-day operation, and various documents that may assist you in learning more about RADIUS.

## 10.3.1 Other RADIUS Servers

There are several available RADIUS servers:

*Cistron RADIUS*

Written by Miquel van Smoorenburg, this server has become widely used in the free-software community. It is completely constructed from the original Livingston source. http://www.radius.cistron.nl

*GNU-radius*

This server is—you guessed it!—another Cistron server-based RADIUS implementation, although unlike the other variants a lot of the code has been rewritten. The server has a rewrite configuration file that is very convenient.

*ICRADIUS*

This server is a variant of the Cistron server. It includes such added features as support for the MySQL database and a front end in interactive HTML. http://radius.innercite.com

*Navis Access*

Lucent's server is an extremely flexible, expandable, and scalable RADIUS server—but it is a commercial product.

*OpenRADIUS*

This server is a completely new implementation with a foundation in the "modular" mind-set, in which all program functionality is based on plug-in code that is completely under the control of the administrator. http://www.xs4all.nl/~evbergen/openradius-index.html

*PerlRADIUS*

This is an effort to write a RADIUS implementation in Perl. This effort seems to be another "me-too" effort: that is, the developers are writing the code merely to say they have written the code. I see no useful benefit from this distribution and, apparently, its development has recently gone on hiatus.

*Radiator*

Another RADIUS server, this is written in Perl and is designed for use in smaller implementations.

*Steel Belted RADIUS*

From Funk Software, this is a commercial product that runs on Windows servers.

*VOP RADIUS*

From VOP Software, this is another commercially-available Windows-based RADIUS server.

*XtRADIUS*

Another Cistron server deviate, XtRADIUS supports extensions for running external programs for accounting or authentication. http://www.xtradius.com

*YARD RADIUS*

This server is derived from the open sources of Livingston RADIUS Server Version 2.1. It has better configuration support and extended features.

## 10.3.2 RADIUS Tools

# Appendix A. Attribute Reference

In this Appendix, the RADIUS standard attributes are listed in order by their attribute number, followed by the official name, the length of the attribute in the packet, and what type of value the attribute supports. Each attribute is then cross-referenced with the main body page explaining the details of the attribute.

Table A-1. The RADIUS standard attributes

| Number | Name | Length | Value | Page |
|---|---|---|---|---|
| 1 | User-Name | 3+ octets | String | User-Name |
| 2 | User-Password | 18-130 | String | User-Password |
| 3 | CHAP-Password | 19 | String | CHAP-Password |
| 4 | NAS-IP-Address | 6 | IP Ad. | NAS-IP-Address |
| 5 | NAS-Port | 6 | Integer | NAS-Port |
| 6 | Service-Type | 6 | Enum | Service-Type |
| 7 | Framed-Protocol | 6 | Enum | Framed-Protocol |
| 8 | Framed-IP-Address | 6 | IP Ad. | Framed-IP-Address |
| 9 | Framed-IP-Netmask | 6 | IP Ad. | Framed-IP-Netmask |
| 10 | Framed-Routing | 6 | Enum | Framed-Routing |
| 11 | Filter-ID | 3+ octets | String | Filter-ID |
| 12 | Framed-MTU | 6 | Integer | Framed-MTU |

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of RADIUS is a Dolium shell. This shell is that of Orcula Dolium, one of a small family of snails. Dolium live in leaf litter or on mossy rocks on mountains such as the Alps and the Carpathians. Their shells are cylindrical, and they have rounded mouths and teeth. Their color varies from yellowish to reddish brown.

Darren Kelly was the production editor and Maureen Dempsey was the copyeditor for RADIUS. Octal Publishing, Inc. provided production services and wrote the index. Sheryl Avruch and Claire Cloutier provided quality control. Interior composition was done by Philip Dangler and Derek Di Matteo.

Hanna Dyer designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Linley Dolby.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

[ Team LiB ]

[A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]



[ Team LiB ]

[ Team LiB ]

[A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]



[ Team LiB ]

keystreams

[ Team LiB ]

[A] [B] [C] [D] [E] [F] [G] [H] [I] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W]



[ Team LiB ]