Automating Linux and Unix System Administration

Second Edition

■ ■ ■

Nate Campi and Kirk Bauer

Apress®

Automating Linux and Unix System Administration, Second Edition

Copyright © 2009 by Nate Campi, Kirk Bauer

ISBN-13 (pbk): 978-1-4302-1059-7

ISBN-13 (electronic): 978-1-4302-1060-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

The source code for this book is available to readers at http://www.apress.com.

I dedicate this book to my dear grandmother Mary Lou.
Her influence makes everyone around her a better person,
and her presence lights up a room.
She is beautiful inside and out,
and she meets adversity with faith,
quiet dignity, and grace.

—Nate Campi

Contents at a Glance

Contents

About the Authors

NATE CAMPI is a UNIX and Linux system administrator by trade, currently working as a UNIX operations manager in San Francisco. His system administration experience is almost entirely with companies with large-scale web operations based on open source software. In his copious free time, he enjoys jogging, watching spaghetti westerns, experimenting with Linux systems, and spending time with his family.

KIRK BAUER has been involved in computer programming since 1985. He has been using and administering UNIX systems since 1994. Although his personal favorite UNIX variant is Linux, he has administered and developed on everything from FreeBSD to Solaris, AIX, and HP-UX. He is the author of various open source solutions such as Logwatch.

Kirk has been involved with software development and system/network administration since his first year at the Georgia Institute of Technology. He has done work for the Georgia Tech Research Institute, Fermi National Accelerator Laboratory, and DHL. In 2000, Kirk was one of the founders and the chief technology officer of TogetherWeb, which was purchased in 2003 by Proficient Systems. Kirk is now a systems engineer with F5 Networks.

Kirk graduated from Georgia Tech in 2001 with a bachelor's degree in computer engineering and is currently pursuing his MBA at Arizona State University. He lives in Peoria, Arizona, with his two dogs, and is looking forward to getting married to his lovely fiancée, Rachel.

About the Technical Reviewer

■**MARK BURGESS** holds a first class honors degree in physics and a Ph.D. in theoretical physics from the University of Newcastle upon Tyne. After working as a physicist, he began to apply the methods of physics to the study of computers and eventually changed research fields to study the formalization of system administration. His current research interests include the behavior of computers as dynamic systems and applying ideas from physics to describe computer behavior. Mark is the author of the popular configuration management software package cfengine. He has received a number of awards including the SAGE 2003 Professional Contribution Award "for groundbreaking work in systems administration theory and individual contributions to the field." He currently holds the Professorship in Network and System Administration at Oslo University College.

Acknowledgments

Only two names are on the book cover, but many talented and dedicated people worked to make this book the best it could be.

We are very grateful to Paul W. Frields from Red Hat for Red Hat Enterprise Linux licenses. This book wouldn't have been possible without them. Mark Burgess lent his unique insight into both cfengine and the book writing process. Our editor Frank Pohlmann is incredibly skilled at finding the weak points in a description and forcing us to explain everything thoroughly. Thanks to our project manager Kylie Johnston; she is a consummate professional. Thanks to our copy editors Nina Goldschlager and Heather Lang, who are very talented and easy to work with. And thanks to our production editor Ellie Fountain.

We really need to thank our families for putting up with our mental absence while writing this book.

Finally, we'd like to thank the energy drink industry for enabling us to stay up late at night even when totally exhausted, go to work the next day feeling like we had been hit by a train, and do it all over again the very next night.

Introduction

> *The system administrator is one of the users of a system, and something more. The administrator wears many hats, as knowledgeable user of UNIX commands, as an operator of system hardware, and as a problem solver. The administrator is also called upon to be an arbitrator in human affairs. A multiuser computer is like a vast imaginary space where many people work and utilize the resources found there. The administrator must be the village elder in this space and settle the disputes that may arise with, hopefully, the wisdom of Solomon.*
>
> —Rebecca Thomas and Rik Farrow
> (*UNIX Administration Guide for System V,*
> Pearson PTR, 1989)

We find it interesting how little UNIX system administration has changed in the last twenty years. If you substitute "computer network" for "multiuser computer," this description still fits perfectly.

The main difference in UNIX system administration between 1989 and 2008 (besides ubiquitous networking) is the sheer number of systems that the average system administrator deals with. Automation is the primary tool to deal with the chaos that can result from so many systems. With it, you can deploy systems identically every time, restore systems to a known good state, and implement changes reliably across all systems (or only an appropriate subset).

We do not claim that the approaches, procedures, and tools used in this book are the only way to set up and maintain a UNIX-based environment. Instead, we walk you through the creation of an example environment, and during the process, help you gain a solid understanding of the basic principles of system automation. This way, you can decide for yourself how you want to set up your own UNIX-based environment.

This book *isn't* like most UNIX/Linux administration books, because it illustrates techniques and principles by building a real UNIX/Linux environment from scratch. We demonstrate that you can configure each host at your site, from installation through production service to system retirement, without logging in and making manual changes to the host. Instead, we'll configure the hosts via imaging systems designed for unattended installation, followed by management with an automation framework.

We wrote this book, because we felt that it is important to demonstrate that an entire site can be managed using automation. Our goal is to be able to quickly, easily, and reliably restore hosts to service after complete system failure. The host might have failed

due to hardware issues; an entire geographic region might be unreachable due to natural disaster, or you might simply have purchased updated hardware on which to run that particular host and need to upgrade. The point of our approach is to configure a host only once and, from that point on, allow an automation system to do that work for you.

Whether you choose to use our exact setup or something completely different, you'll have gained knowledge and experience by going though the process with us in our example environment. Our promise to you is that if you need to configure a new UNIX-based infrastructure from scratch (and you're able or allowed to use the operating systems and software we demonstrate), you can use this book to create a fully functional and scalable new infrastructure. Every service and piece of architecture that our new environment needs is set up using automation.

This book moves fast and will be best utilized if you follow along with the examples and implement the described steps on systems of your own. In addition, download the code and configuration files from the Source Code page of the Apress web site (`http://www.apress.com`).

Who This Book Is For

This book is written for the experienced system administrator. We have made every attempt to refer you to appropriate external sources when we weren't able to delve into great detail on a service or protocol that we were automating. In addition, little explanation is given to the usage of basic UNIX/Linux commands and shell scripts. You don't, however, have to be an advanced system administrator. We feel that a system administrator with only one or two years of full-time on-the-job experience is more than ready to utilize the concepts and tools in this book.

How This Book Is Structured

The book begins with four introductory chapters that you should be very familiar with before you move on to later, more detailed chapters. The later chapters, starting with Chapter 5, build a new UNIX environment: we set up an automation system; automate installation systems; and enhance the site with real applications, monitoring, reporting, and security.

Chapter 1, "Introducing the Basics of Automation," covers the reasons for and benefits of automation, as well as the methodology behind it. Also, the `sudo` utility is introduced and explained.

Chapter 2, "Applying Practical Automation," covers the steps behind automating a common procedure—adding a new user account. During the process, the core tenets of automation are covered.

Chapter 3, "Using SSH to Automate System Administration Securely," covers the basics of using secure shell (SSH), discusses SSH security concerns, describes how to set up public key authentication in SSH, and delves into various other related topics such as SSH log analysis.

Chapter 4, "Configuring Systems with cfengine," explains the concepts behind cfengine, as well as the various cfengine daemons and utilities. A full discussion takes place of the common configuration settings in the main cfengine configuration file. The requirements for a minimal cfengine architecture with two hosts are fully explored.

Chapter 5, "Bootstrapping a New Infrastructure," covers the cfengine configuration for a new, automated UNIX/Linux environment. A "master" cfengine host is set up, with all the required configuration files to manage new Red Hat Linux, Debian Linux, and Solaris hosts. This is the first step in building a UNIX/Linux environment from scratch using automation.

Chapter 6, "Setting Up Automated Installation," demonstrates the automated installation of Red Hat Linux using Kickstart, Debian Linux using Fully Automatic Installation (FAI), and Sun Solaris using Jumpstart. The hosts deployed in this chapter continue to be used in the later development of our example UNIX/Linux infrastructure.

Chapter 7, "Automating a New System Infrastructure," covers the automation of these services and procedures in our new infrastructure: the Network Time Protocol (NTP), Domain Name System (DNS), standardized local account files and new user accounts, mail routing, and home directories mounted with the Network File System (NFS).

Chapter 8, "Deploying Your First Application," covers the deployment and configuration of the Apache web server, demonstrating various ways to automate the distribution of both the web server daemon binaries and the web content. Along the way, you learn about sharing data with NFS, rsync, scp, cfengine data copies, and Subversion.

Chapter 9, "Generating Reports and Analyzing Logs," covers automated syslog and cfengine log analysis and reporting in our new infrastructure.

Chapter 10, "Monitoring," uses cfengine to automate the deployment and configuration of Ganglia and Nagios in our example environment.

Chapter 11, "Infrastructure Enhancement," uses cfengine to manage version control with Subversion, including branching the cfengine configuration tree to create testing and development environments. Also, backups are handled, in a very simple way.

Chapter 12, "Improving System Security," covers the implementation of security enhancements with cfengine. Measures undertaken include removing the SUID bit from root-owned binaries, protecting system accounts, applying UNIX/Linux patches and vendor updates, shutting down unneeded daemons, adding host-based firewalls, and more.

Appendix A, "Introducing the Basic Tools," provides a basic introduction to the tools used throughout this book and provides a good starting point for understanding and utilizing the examples presented in this text. This appendix covers the following tools: bash, Perl, grep, sed, and AWK.

Appendix B, "Writing cfengine Modules," covers extending cfengine through modules. This is a quick but thorough introduction using examples.

Downloading the Code

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section of this book's home page. Please feel free to visit the Apress web site and download all the code there. You can also check for errata and find related titles from Apress.

Contacting the Authors

We have gone through several stages of proofreading and error checking during the production of this book in an effort to reduce the number of errors. We have also tried to make the examples and the explanations as clear as possible.

There may, however, still be errors and unclear areas in this book. If you have questions or find any of these errors, please feel free to contact us at `nate@campin.net`. You can also visit the Apress web site at `http://www.apress.com` to download code from the book and see any available errata.

■ ■ ■

Introducing the Basics of Automation

When one of this book's authors was in high school, he got his first part-time job keeping some of the school's computers running. He loved it. He did everything by hand. And because the school had only two or three computers, doing everything by hand wasn't a big issue. But even then, as the number of systems grew to five, six, and finally more than ten, he realized just how much time you can spend doing the same things over and over again. This is how his love of automation was born.

This book's other author found automation through necessity as well, although later in his career. During the so-called "tech downturn" around the year 2003 in Silicon Valley, he suddenly found himself the sole member of what had been a three-person system-administration team. The number of systems and responsibilities were increasing, while staffing levels had dramatically decreased. This is when he found the cfengine automation framework. Cfengine drastically reduced the amount of time required to implement system changes, allowing him to focus on improving the infrastructure instead.

In this chapter you will learn the basics of automating system administration so that you can begin to make your life easier—as well as the lives of everybody who uses or depends on your systems. The topics covered in this book apply to a wide variety of situations. Whether you have thousands of isolated systems (sold to your customers, for example), a large number of diverse machines (at a large company or university campus), or just a few servers in your home or small business, the techniques we'll cover will save you time and make you a better administrator.

Throughout this book, we will assume the reader has a basic set of UNIX skills and some prior experience as a system administrator (SA). We will use numerous tools throughout the book to provide example automation solutions. These tools include the following:

- The Bash shell

- Perl

- Cfengine

- Regular expressions

- The `grep` command

- The `sed` stream editor

- AWK

If you are not familiar with one or more of these tools, read their introductions in the Appendix before you proceed. See Chapter 4 for an introduction to cfengine.

Do You Need Automation?

If you have one Linux system sitting on your desk at home, you don't *need* automation. You can take care of everything manually—and many people do. But you might *want* automation anyway because it will ensure your system has the following characteristics:

- *Routine tasks such as performing backups and applying security updates take place as scheduled*: This saves the user time and ensures that important tasks aren't forgotten.

- *The system is consistently set up*: You might have one system, but how often is it replaced due to faulty hardware or upgrades? When the system hardware is upgraded or replaced, an automation system will configure the software again in the same manner as before.

- *The system can be expertly configured, even if you're not an expert*: If you use automation built by someone more experienced with system configuration and automation, you benefit from his or her expertise. For example, you benefit from the Red Hat Network (RHN) when using a licensed installation of Red Hat Enterprise Linux. RHN regularly supplies automated software updates that are reliable and timely, resulting in a more secure and stable system. Most users don't have the required system configuration and programming skills to implement such a system, so Red Hat developed a solution that any of their software licensees can use freely.

- *The system is in compliance with guidelines and standards*: You might be responsible for only one system, but if the system belongs to your employer, it might be subject to regulatory or other legislative requirements around security and configuration. If this is the case, an automation system that enforces those requirements supplies the documentation needed to prove compliance. Even if no laws or credit

card–company guidelines apply, your employer might require that all systems on its network meet certain minimal security standards. Usually a one-time manual configuration isn't enough to satisfy these standards; an automated solution is required.

- *The system is reliable*: If solutions to occasional problems are automated, the system is more reliable. When a disk fills up with temporary files, for example, the user who employs an automation system can schedule a daily cleanup procedure to prevent failed writes to disk and system crashes from full disks.

Likewise, you might think you don't need automation if you have only one server in your company. However, you might want it because backups and timely security updates are easy tasks for a busy system administrator to neglect, even in this most basic setup. In addition, if your company's server is a file server or mail server, its drives will tend to fill up and cause problems. In fact, any security or stability problem with this type of computer will likely result in expenses for the company, and any loss of data could be disastrous. This is exactly the reason OS vendors rotate the log files for the daemons they install on the system, because they know the end result of unmaintained log files. An automation system can also help out your successor or the person covering for you during your vacation.

When it comes down to it, the number of machines isn't an important factor in the decision to use automation. Think of automation as insurance that the machine is being monitored. A Red Hat Package Manager (RPM) install or security update can undo a manual change to a configuration file, for example. If an automation system enforces the policy that the configuration file contains a particular entry or value, it will reapply the change if necessary.

In addition to log-file rotation, your OS distributor already automates many tasks on a stand-alone system. It makes security checks, updates databases with information on file locations (e.g., `slocate`), and collects system accounting and performance information. All this and more happens quietly and automatically from within a standard UNIX or Linux system.

Automation is already a core part of UNIX philosophy, and cron jobs have historically been the de facto method for automating UNIX tasks. In this book we favor cfengine for task automation, but for now you can think of cfengine as a next-generation cron daemon.

For the sake of the single system, it's fine to go the simple route. You can add more log-rotation settings to already automated systems such as the "logrotate" utility (standard on all Linux distributions that we can think of). You don't need something complex, but you do need automation if you want to ensure important tasks happen regularly and reliably.

You should do everything you can to prevent problems before they happen. If you can't do that, follow the advice of one of our old managers: make sure the same problem

never happens again. If a disk fills, set up a log-rotation script run from cron that deletes unneeded temporary files—whatever addresses the root cause. If a process dies, set up a process monitor to restart it when it exits. In later chapters, we will show you how to accomplish these tasks using cfengine. The automation systems at most sites grow over time in response to new issues that arise.

SAs who respond to all problems with permanent (read: automated) solutions go a long way toward increasing overall availability of their sites' applications and services. Automated solutions also allow them to get some sleep while on call. (The sleep factor alone is reason enough for most SAs to spend a lot of time on automation.)

So, back to the question—do you *need* automation? We'll introduce a variety of situations that require automation and discuss them further throughout the book.

Large Companies with Many Diverse Systems

The most traditional situation requiring automation involves a large company or organization with hundreds or even thousands of systems. These systems range from web servers to file servers to desktop workstations. In such a situation, you tend to have numerous administrators and thousands of users.

You might treat the systems as several groups of specialized servers (i.e., all workstations in one group, all web servers in another) or you might administer all of them together. Either way, with a large number of different systems, automation is the only option. Cfengine is especially suited to this type of environment. It uses a high-level configuration file and allows each system to pull its configuration from the configuration server. One of cfengine's key strengths: Not only can it configure hundreds or even thousands of systems in exactly the same manner, but it can also configure a single system in a unique way. We'll discuss cfengine thoroughly in later chapters.

Medium-Sized Companies Planning for Growth

Any medium-sized or small company is in just about the same situation as the large companies. You might have only 50 servers now and some basic solutions might work for you, but you probably hope to expand. Automation systems built on cfengine scale from a few systems to many thousands of systems. The example cfengine infrastructure demonstrated in Chapter 5 assists scalability by segmenting the configuration into many files. Sites with more than 25,000 hosts use cfengine.

You might have only one type of a particular system, but if it fails, cfengine can reproduce the original system quickly and reliably. Normally at that point some user or application data needs to be restored, but that's much easier than reproducing a system from a base install.

Internet Service Providers

If you work at an Internet Service Provider (ISP), you probably have more computers than employees. You also (hopefully) have a large number of customers who pay you money for the service you provide. Your systems might offer a wide variety of services, and you need to keep them all running. Other types of companies have some critical servers, but most of their systems are not critical for the companies' success (e.g., individual workstations, testing systems, and so on). At an ISP, almost all of your systems are critical, so you need to create an automation system that promotes system stability and availability.

Application Service Providers

If you're an application service provider (ASP), you might have hundreds of systems that all work together or numerous groups of independent systems. Your system-administration tasks probably include deploying and configuring complex, custom software. You must synchronize such changes among the various systems and make them happen only on demand. Stability is very important, and by minimizing changes you can minimize downtime. You might have a central administration system or a separate administration for each group of systems (or both). When you create your automation system, be sure to keep an eye on scalability—how many systems do you have now, and how many will you have in the future?

Fortunately with cfengine you already have an automation system; what you need to keep in mind is that in such an environment you often need additional capacity in a hurry. Being able to boot new hardware off the network and have cfengine configure it appropriately means that the most time-consuming aspect of adding new systems is the time required to order, rack, and cable up the new systems. This is the ideal situation for an ASP, and the SA staff in such shops should aspire to it.

Web Server Farms

Automation within web clusters is common today. If you have only a couple of load balancers and a farm of web servers behind them, all your systems will be virtually identical. This makes things easier because you can focus your efforts on scalability and reliability without needing to support differing types of systems. In a more advanced situation, you also have database systems, back-end servers, and other systems. In this case, you need a more flexible automation system, such as cfengine. Regardless of the underlying infrastructure, web servers will be plentiful. You need a quick and efficient way to install and configure new systems (for expansion and recovery from failures). Sound familiar? The core needs and considerations are common across different business types. We'll return to these recurring themes at the end of the chapter.

Beowulf Clusters

Beowulf clusters are large groups of Linux systems that can perform certain tasks on par with a traditional supercomputer. Regardless of whether you use a Beowulf cluster or another type of computational cluster, each cluster usually has one control system and hundreds of computational units. To set up and maintain the cluster efficiently, you need the ability to install new systems with little or no interaction. You have a set of identical systems, which makes configuration easy. You also usually have maintenance periods during which you can do what you want on the systems, which is always nice. But when the systems are in use, making changes to them might be disastrous. For this reason, you will usually want to control the times when the systems will accept modifications.

Hosts in such clusters will typically boot off the network and load up a minimal operating system entirely into memory. Any local storage on the system is probably for application data and temporary storage. Many of the network boot schemes like this completely ignore the containment of system drift during the time between boot and shutdown.

In a worst-case scenario, an attacker might access the system and modify running processes, access other parts of your network from there, or launch attacks against other sites. A less extreme problem would be one where certain applications need to be kept running or be restarted if they consume more than a defined amount of memory. An automation system that ignores the need to control a running system is at best only half an automation system. Using a system reboot to restore a known good state is sufficient if the site administrators don't wish to do any investigation or improvement. A reboot is only a temporary solution to a system problem. An attacker will simply come back using the same mechanism as before, or processes will still die or grow too large after a reboot. You need a permanent solution.

A cluster designed to network-boot can just as easily run cfengine and use it to contain system drift. You'll find helpful cfengine features that can checksum security-critical files against a known good copy and alert administrators to modifications. Other cfengine features can kill processes that shouldn't be running or restart daemons that are functioning incorrectly. Systems that are booted from identical boot media don't always have the same runtime behavior, and cfengine allows you to control the runtime characteristics of your systems.

For some of the best documentation on system drift and ways to control it, check out the book *Principles of Network and System Administration, Second Edition* by Mark Burgess (Wiley, 2004). The author approaches the subject from an academic standpoint, but don't let that scare you away. He uses real-world examples to illustrate his points, which can pay off at your site by helping you understand the reasons behind system drift. The book will help you minimize these negative effects in your system and application design.

Network Appliances

Finally, many companies produce what we call "network appliances," which are systems that run some UNIX variant (often Linux or FreeBSD) and are sold to customers as a "drop-in" solution. Some current examples of these products include load balancers and search engines. The end user administers the systems but might know very little about UNIX. End users also usually do not have `root` access to the system. For this reason, the system must be able to take care of itself, performing maintenance and fixing problems automatically. It will also need to have a good user interface (usually web-based) that allows the customer to configure its behavior. Such vendors can leverage cfengine so that they can focus on their core competency without worrying about writing new code to keep processes running or file permissions correct.

What Will You Gain?

The day-to-day work of system administration becomes easier with automation. We can promise the following benefits, based on our own experience.

Saving Time

You can measure the time saved by automation in two ways. The first is in the elapsed wall-clock time between the start and end of a task. This is important, but not as important as the amount of actual SA time required. If the only SA time required is in setting up the task to be automated in the first place and occasionally updating the automation from time to time, the benefits are much greater than faster initial completion. This frees the SA to work on automating more tasks, testing out new software, giving security or reliability lectures to the in-house programmers, or simply keeping current with recent technology news.

Reducing Errors

Unfortunately, you'll see a rather large difference between systems built according to documentation and systems configured entirely through automated means. If you were to audit two systems for differences at a site where all systems were configured by cfengine, the differences should—in theory—arise only from errors outside the automation system, such as a full disk. We know from firsthand experience that systems configured according to a written configuration guide invariably differ from one another. After all, humans are fallible. We make mistakes.

You can reduce errors at your site by carefully testing automated changes in a non-production environment first. When the testing environment is configured properly, only then do you implement the change in your production environment.

For the sake of this book, the term "production" means the systems upon which the business relies, in any manner. If the company is staffed primarily with nontechnical people, perhaps only the SA staff understands the differentiation when the term is used. Trust us, though: the business people understand when particular hosts are important to the business and will speak out about perceived problems with those systems.

Documenting System Configuration Policies

Whether the automated configuration at a site is done by shell scripts, Perl scripts, or a tool such as cfengine, the automation serves as documentation. It is in fact some of the most usable documentation for a fellow SA, simply because it is authoritative.

If new SAs at a site read some internal documentation about installing and configuring some software, they don't have any assurance that following the documentation will achieve the desired effect. The SA is much better off using a script that has been used all the previous times the software needed to be installed and configured.

Either the script will work and the proper results will emerge, or it'll break because of some change in the environment. The change should be much easier to find based on error output from the script. If the steps on a wiki page or a hard copy of the documentation don't work, on the other hand, the error could be due to typos in the doc, steps omitted, or updates to the procedure not making it back into the docs. Using automation instead helps insulate the SA against these scenarios.

Realizing Other Benefits

This book applies to a wide range of people and situations, so not all the material will be of interest to all readers. If you haven't yet created an automation system or implemented an open source framework (such as cfengine) from scratch, this book will show you how to get started and how to take the system from initial implementation through full site automation. You will also learn the principles that should guide you in your quest for automation. As your skills and experience grow, you will become more interested in some of the more advanced topics the book discusses and find that it points you in the right direction on related subjects.

If you already have an automation system of some sort, this book will provide you with ideas on how to expand it. There are so many ways to perform any given task that you are sure to encounter new possibilities. In many cases, your current system will be advanced enough to leave as is. In other cases, though, you will find new ways to automate old tasks and you'll find new tasks that you might never have considered automating.

Don't write off a complicated manual task as too difficult to automate before carefully evaluating the decisions made during the process. You'll usually find during manual inspection that the decision process is based on attributes of the system that cfengine or a script can collect. The act of documenting a change before making it usually forces the SA to approach the problem in a systematic way. The change process will end up producing better results when the process is planned this way.

Imagine that you often have to restart a web-server process on one of your servers, in a sequence of actions such as this:

- You check a log file for a commonly recurring error message.

- You check if CPU utilization is high.

- You test the web server using a command-line utility, looking for a successful HTTP status message.

You can collect each of these manual checks automatically, and a script or cfengine can make the decision to restart. If this makes you nervous, write the script's collection aspects first, and at the point where a system change would be made, instruct the script to print a message to the screen about the decision it has reached. Run the script, then manually go through your decision process independently of the script. Enhance the script each time its decision differs from yours. You'd be surprised at the complex procedures you can automate this way. You don't have to enable the automated restart itself until you're comfortable that it will do the right thing.

AUTOMATING A DIFFICULT PROBLEM/RESPONSE PROCEDURE

One of us works at a site where the SA staff used complex manual procedures to fix a distributed cluster when application errors would occur. The manual process would often take several hours to completely restore the cluster to a working state.

The staff slowly automated the process, beginning with simple commands in a shell script to avoid repeatedly typing the same commands. Over time the staff enhanced the script with tests to determine which errors were occurring and to describe the state of the cluster's various systems. Based on these tests, the script could determine and perform the correct fix.

Eventually, the SA staff used the automated process to repair the cluster in as little as a few minutes. In addition, the script incorporated so many of the decisions previously made by the SA staff members that it became the foremost authority on how to deal with the situation. Essentially, the script serves as documentation on how to deal with multiple issues and situations on that particular application cluster.

When it comes to computer systems, every environment is different—each has different requirements and many unique situations. Instead of attempting to provide the unattainable "one solution fits all," this book shows how to set up an example environment. As we configure our example environment, we will explain the decision process behind the solutions we have chosen. After you have learned these options, you will be able to make an informed choice about what you should automate in your environment and how you should do it.

What Do System Administrators Do?

Life as a system administrator usually falls into three categories:

- Tedious, repetitive tasks (a.k.a. boring tasks)

- New, innovative tasks (a.k.a. why you love the job)

- Answering users' questions, or otherwise dealing with monitoring alarms, issues or emergencies (a.k.a. pulling your hair out)

The goal of this book is to help you create new and innovative solutions to eliminate those tedious and repetitive tasks. And if you find a way to automate the task of answering users' questions, please let us know! But even if you can't, you can at least create a system that detects and even fixes many problems before they come to the attention of the users, or more important, your monitoring systems. Also, any task you have automated is a task the users could potentially perform on their own.

System administrators spend time on other tasks, of course, but we won't address them here because they aren't pertinent to this discussion. (These might include browsing the Slashdot web site, checking on reservations for the next science-fiction convention, or discussing a ham-radio setup with other geeks around the office.) Suffice it to say that following the guidelines in this book will allow you to spend more time on these other tasks and less time on the tedious tasks and emergencies.

You can classify the tedious tasks into the following categories:

- *Preinstallation*: Assigning an IP address, configuring existing servers and network services, and so on

- *Installation*: Installing a new operating system and preparing it for automation

- *Configuration*: Performing initial configuration and reconfiguration tasks

- *Managing data*: Duplicating or sharing data (users' home directories, common scripts, web content, etc.), backups, and restores

- *Maintenance and changes*: Rotating logs, adding accounts, and so on

- *Installing/upgrading software*: Using package management and/or custom distribution methods

- *System monitoring and security*: Performing log analysis and security scans; monitoring system load, disk space, drive failures, and so on

Methodology: Get It Right from the Start!

Automating tasks proves much more useful when you apply a consistent methodology. Not only will you have less direct work (by having code that is easier to maintain and reuse), but you will also save yourself and others time in the future. Whenever possible, we'll include techniques in this book that support these basic methodologies:

- Activities you have performed must be reproducible.

- Any system's state must be verifiable.

- Problems should be detected as they occur.

- Problems should be repaired automatically, if possible.

- The automation methods must be secure.

- The system should be documented and easy to understand.

- Changes should be testable in a safe environment.

- Every system change should be examined for side effects that also must be handled automatically.

Perhaps the most important aspect of any automated system is reproducibility. If you have two machines configured just the way you like them, you should be able to add an identically configured third machine to the group with minimal effort. If somebody makes an incorrect change or loses a file, restoring the system to full functionality should be relatively easy. These nice capabilities all require that you can quickly and perfectly reproduce what you have done in the past or to other systems. Even if you don't plan to add more systems, you can bet that at some point one of your systems will fail. It might be the CPU or disk(s), or you might have a fire in your server room. (You do have a disaster recovery plan, right?) The experienced SA protects his systems against their inevitable failure, and automation is a big part of the solution.

You also need to be able to verify a system's status. Does it have the latest security updates? Is it configured correctly? Are the drives being monitored? Is it using your newest automation scripts, or old ones? These are all important questions, and you should be able to easily determine the answers if your automation system is implemented properly.

In many cases, detecting problems is a great step forward in your automation process. But how about automatically fixing problems? This too can be a powerful technique. If systems fix their own problems, you will get more full nights of sleep. But if your auto-repair methods are overzealous, you might end up causing more problems than you solve. We will definitely explore self-repair whenever appropriate.

An administrator always has to consider security. With every solution you implement, you must be certain you are not introducing any new security issues. Ideally, you want to create solutions that minimize or even eliminate existing security concerns. For example, you might find it convenient to set up Secure Shell (SSH) so that it uses private keys without a passphrase, but doing so usually opens up serious security holes.

There will always be people who follow in your footsteps. If you ask them, the most important component of your work is good documentation. We already mentioned that in many cases automation techniques provide automatic documentation. You should take full advantage of this easy documentation whenever possible. Consider, as an example, a web server under your control. You can manually configure the web server and document the process for yourself and others in the future, or you can write a script to configure the web server for you. With a script, you can't neglect anything—if you forget to do something, the web server does not run properly.

As obvious as it might sound, it is important to test out your automation before you deploy it on production servers. One or more staging machines are a must. We will discuss techniques for propagating code across machines and explain how you can use these techniques for pushing code to your staging server(s).

Whenever you automate a task, you must consider dependencies. If you automated the installation of software updates and Apache is automatically upgraded on your systems, that's great. But if the configuration files are replaced in the process, will they be regenerated automatically? You need to ask yourself these kinds of questions when you automate a task.

What do you do about these dependencies? They should be your next project. If you can automatically upgrade but can't automatically configure Apache, you might want to address that task next. Even if you have already automated this task, you need to make sure the automation event is triggered after the software is updated. You might also need to update a binary checksum database or services on your systems. Whether or not these tasks are automated, you need to be sure they will not be forgotten.

Homogenizing Your Systems

Most people reading this book will have a variety of UNIX systems within their network. If you're lucky, they will all run the exact same operating system. In most cases, though, you will have different systems because there are a wide variety of commercial UNIX systems as well as FreeBSD and Linux. Even with one type of UNIX, you might have different varieties (called "distributions" in Linux). Even if all your systems run the same UNIX system, some might run older versions than others.

The more similar your systems, the better. Sure, you can have a script that behaves differently on each type of system. You can also use classes in cfengine to perform different actions on different systems (discussed throughout the book). These approaches will be necessary to some degree, but your first and best option is to minimize these differences among your systems.

Your first step: Provide a certain base set of commands that operate the same way on all systems. The GNU Project (`http://www.gnu.org`) is helpful because the GNU developers have created open source versions of most standard UNIX commands. You can compile these to run on any system, but most of them are binary programs, so you'll need to compile each program for each platform or find prebuilt packages. You can then distribute these programs using the methods discussed in Chapter 8. Once they reside on all your systems in a standard location (such as `/usr/local/`), you should use them in all your scripts.

Some operating systems will provide other helpful commands that you might want to have on all your systems. If you're lucky, these commands will be shell or Perl scripts that you can modify to operate on other systems. Even if they are binary commands, they might be open source and therefore usable on commercial UNIX systems.

In addition to consistent commands, a consistent filesystem layout can be helpful. As we already mentioned, placing custom commands in the same location on all systems is a must. But what else is different? Do some of your systems place logs in `/var/adm/` and others in `/var/log/`? If so, you can easily fix this with symbolic links.

We recommend that you consider each difference separately. If it is easy to modify your systems to make them similar, then do so. Otherwise, you might be able to work around the differences, which is what you should do. Finally, if it isn't too difficult to add a specific set of consistent commands to all your systems, try that approach. In most cases, you will have to use some combination of all three of these approaches in your environment.

Deciding on Push vs. Pull

You can take one of two main approaches when configuring, maintaining, and modifying systems: the "push" method or the "pull" method. The "push" method is when you have one or more systems contact the rest of the systems and perform the necessary tasks.

You implement the "pull" method by having the systems contact one or more servers on a regular basis to receive configuration instructions and configure themselves. Both methods have their advantages and disadvantages. As usual, the one you should choose depends on your situation. We personally have a favorite, but read on as we present the options.

The push method gives the SA the feeling of control, because changes are triggered actively by one or more systems. This scenario allows you to automatically configure, update, or modify your systems, but only when you (or some other trigger) cause it to happen.

The push method sounds great, right? Well, not exactly—there are plenty of drawbacks. For instance, what if you have more than 1,000 systems? How long would it take to contact every system when you need to make a change? What happens if some systems are currently unavailable? Are they just forgotten?

This is where the pull method really shines. If you make a change to one or more configuration servers, all your systems will pick up those changes when they can. If a system is a laptop at somebody's home, it might not get the changes until the next day. If a system has hardware problems, it might not get the changes until the next week. But all your systems will eventually have the changes applied—and most almost immediately.

So, does your environment consist of several systems that are intricately related? Do these systems need to be updated and modified together at all times? Does the update process unavoidably cause some amount of service outage? If so, you probably want to push any changes to these systems. If these aren't issues for you, and especially if you have a large number of systems, then the pull method is generally preferable.

Regardless of the method you choose, you still must be aware of the loads that will be placed on your systems, your network, and especially your servers. If you push in series (one system at a time), you are probably okay. But if you push in parallel (all systems at once), the server might suffer. If your clients pull from a server, be sure they don't all pull at the same time. Consider adding a random delay before the task begins. Cfengine, which uses the pull method, provides the `SplayTime` option that does just this.

Dealing with Users and Administrators

Everyone who uses your systems is either a user or an administrator (where an administrator is usually a user as well). At an ISP, most employees are administrators but the customers are actually the users. At a traditional company, a small number of people are administrators and all other employees are users.

Your more technical users might also be administrators of their own desktop systems. These systems can still be security risks, so you should include them in your automation system. You have to be aware of conflicts that might arise between your automation system and the user's own actions. The user might destroy something your system did, in

which case the system should do it again automatically. Similarly, your automation might destroy changes the user wanted to make on his or her system—you would have to work with the user to find a different way to make the change.

What you have to worry about the most are any interactions that might cause problems with the system. If, for example, your automation system assumes that a certain account resides on the system, it might not operate without it. This isn't a problem—unless, of course, somebody manually deletes that user.

Ideally, you would have a list of every assumption your automation system makes about every system. You would then enhance your automation system to check all these assumptions and repair any problems. Realistically, you would have a hard time reaching this ideal, but the more hands you have in the pot (i.e., the more administrators), the harder you should try.

Another concern, if you have more than one or two administrators for a system, is an audit trail. Who has been accessing each system and what have they been doing? Most systems provide process accounting—a log of every executed process, the user who executed it, and the amount of time it was running. You usually have to enable this logging because it can consume quite a bit of drive space.

The problem is that when you see that `root` executed the command `rm -rf /home/*`, how do you know *who* did it? You know that the `root` user ran it, but who was logged in as `root` at that time? Did you make an unfortunate typo, or did the pissed-off employee who quit yesterday do it on purpose?

The easiest solution when you have multiple administrators is to give the `root` password to everybody, but this provides no audit trail at all. A better option is to specify which SSH keys should provide access to the `root` account. Each user has his or her own private SSH key and, assuming the logging is turned up slightly, the SSH server records the key used for each login. This allows you to determine who was logged in as `root` at any given time. You can find information on this approach in Chapter 3.

There is still a chance that multiple people will be logged in as `root` when a problem has occurred. The only way to know exactly who ran which commands is to use Sudo. Sudo is a program that allows specified users (or any user, really) to execute specified commands as `root`. Using it is easy:

```
kirk % sudo /etc/init.d/httpd start
Password:
Starting httpd:                                    [  OK  ]
```

Note that Sudo prompts you for a password. It wants you to enter the password for your user account, not the `root` account. This request helps verify that the person using the `kirk` account is still Kirk. The authentication will last for some period of time (usually five minutes) or until the command `sudo -k` is executed.

Executing that command as kirk results in the following log entry (sent through sys-log, which ends up in /var/log/secure on our system):

```
kirk : TTY=pts/13 ; PWD=/tmp ; USER=root ; COMMAND=/etc/init.d/httpd start.
```

■**Note** You can find the code samples for this chapter in the Downloads section of the Apress web site (http://www.apress.com).

None of this will work, however, without the proper permissions in the Sudo con-figuration file: /etc/sudoers. You can edit this file manually, but if more than one person might edit the file at the same time, you should use the visudo command. This command also checks the file for valid syntax on exit.

Here is the entry that allows kirk to start the web server:

```
kirk ALL = /etc/init.d/httpd start
```

This line says that the user kirk is allowed, on any host (ALL), to run the command /etc/init.d/httpd start. You could also allow the web server to be stopped and restarted by allowing any parameter to be specified to this script:

```
kirk ALL = /etc/init.d/httpd
```

You can also limit this command so that it can be executed only on the web server:

```
kirk www = /etc/init.d/httpd
```

This would allow the same /etc/sudoers file to be used on all of your systems (if this is the way you want it). You can even allow certain users to execute commands as other specific users:

```
kirk www = (nobody) ls
```

This allows kirk to list directories as the user nobody. You might find this useful for verifying permissions within web content. If you can list directories with this command, the web server can also get into the directory. You could also apply this rule to all users in a specific group:

```
%users www = (nobody) ls
```

This command allows anybody in the group `users` to execute the command `ls` (with any arguments) as the user `nobody` on the host `www`. You could even remove the password prompt as well:

```
%users www = (nobody) NOPASSWD: ls
```

Now the users won't have to enter their passwords at all when they run this command. Because this command isn't that dangerous in most cases, removing the password requirement is a nice option.

With Sudo, you can run certain commands without a password to allow scripts that are running as a user other than `root` to execute system commands. This is the most beneficial way to use Sudo when it comes to automation.

■**Warning** It might be tempting to provide unlimited `root` access to certain users through Sudo. Although this will allow the users to execute commands as `root` with full logging enabled, it is not usually the most secure thing to do. Because each user can run commands as `root` with his or her `user` password, you effectively have several `root` passwords for the system.

Many more options are available to you within the `/etc/sudoers` file. We're not going to attempt to cover them here, but you can view the `sudo` and `sudoers` man pages as well as the `http://www.courtesan.com/sudo/` web site for more information.

Who Owns the Systems?

The systems and services on your network aren't yours to change at will. Normally your company has established people empowered to make business decisions about when a service can and should go down for maintenance. These staff members understand the established requirements for advance notifications to customers, partners, and users. They usually also understand internal or external factors that would affect whether a scheduled time is a good fit for the business.

You can't decide on your own to make changes in an unannounced window, or perform maintenance that takes down some functionality of your applications or systems without prior approval. You need to schedule downtime and/or changes that affect or might affect production services with your stakeholders. The SA might very well be the person empowered to make the decision, but then the SA needs to communicate the activity with enough advance notice to satisfy any internal or external SLAs (Service Level Agreements).

This information is probably well known to most readers, but a reminder is useful even to advanced SAs. SAs often get very close to their systems and applications, so they might forget that the decisions about what's best for their systems don't start and stop with them.

Defining Policy

We keep mentioning "policy," which might sound like a big document handed down from on high, bound in leather and signed in blood by all executives at your company. This isn't what we mean. The configuration policy is highly technical, and although it's influenced by factors outside the technology team (i.e., legislation, credit card–security guidelines, site security policy, and so on), it is purely a statement of how the SA team believes the systems should be configured.

The problem with most sites (whether running UNIX-like operating systems, Windows, or other OSs) is that many machines will at best only partially comply with policy. All systems might be imaged exactly the same way, but over time user and SA activities make enough changes to each host that the system drifts from the desired state.

Sites that use automation for all aspects of system configuration will still suffer from some drift associated with users and networked applications. Examples of this drift include varying disk utilization based on log files from daemons or files left on the system by users, or stray processes left around by users. This should be the extent of the drift, because the automation system should install and configure all configuration files and programs, as well as keep them in conformance with policy. In addition, as drift is observed, you can update the automation system to rein in its effects.

You already have a system configuration policy, but there's a good chance that it's documented incompletely. There's an even better chance that some or all of it exists only in your head. This book exists so that you can move it from wetware into software.

CHAPTER 2

■ ■ ■

Applying Practical Automation

You need to know several key things before you automate a new procedure or task. (Well, first you need to know where your soda and potato chips are. Find them? Okay, moving on.) This chapter presents the prerequisite information in an easy-to-digest format. We'll demonstrate these same key points in later chapters when configuring our example systems. You might want to review this chapter after reading the entire book, especially when embarking on a new automation project.

This chapter assumes familiarity with Bourne Shell scripting. Experienced SAs shy away from scripting specifically for the Bash shell (Bourne-Again SHell) except when absolutely necessary. Even if your site has Bash installed everywhere today, you might have to integrate some new systems into your infrastructure tomorrow due to an acquisition. If the script that does some initial automation framework setup—such as installing cfengine or other required administrative utilities—doesn't work on the new systems, you're in for some serious extra work. If your scripting is as portable as possible from the start, in effect you're buying insurance against future pain.

Seeing Everything As a File

One of the core strengths of UNIX and UNIX-like operating systems is the fact that almost everything on the system is represented to the user as a file. Both real and pseudo devices (such as `/dev/null`, `/dev/zero`, and so on) can be read from and (often) written to as normal files. This capability has made many operations easy, when the same results would be difficult to attain under other operating systems. Be thankful for the UNIX heritage of being written for and by programmers.

For example, if you want to create an ISO file on a remote system from a DVD in your laptop, you could run this:

```
dd if=/dev/cdrom | ssh remotehost 'dd of=/opt/big/vmware/sol10.iso'
```

Linux represents the CD/DVD drive as a file, in this case `/dev/cdrom`, so you simply use the `dd` command to copy it bit for bit to a different file. If you don't have the disk space on your laptop for storing the ISO file, you can pipe the `dd` output over SSH and use `dd` again on the remote host to place the output in a single file.

You can then configure VMware to mount the ISO file as a CD-ROM drive (standard VMware functionality) and quickly boot from the device and install on a host with no physical CD/DVD drive.

You probably won't ever need to automate ISO-file creation (although every site is different), but it's important to remember that *the vast majority of automation operations are based on copying and/or modifying files.* Either you need to update a file by copying a new file over it, edit the file in place, or copy out an additional file or files.

Often when files change or new files are distributed, a process on the host needs to restart so the host can recognize the change. Sometimes a host process starts for the first time if the new files comprise a new server/daemon process distributed as a package, tar-ball, or simply a file.

The bulk of what we'll be doing in this book is copying files, modifying files, and taking actions based on the success or failure of earlier file operations. Certain operations might prove tricky, but most of what we're doing should be familiar to UNIX SAs.

Understanding the Procedure Before Automating It

We've seen many administrators open a cfengine config file to automate a task and end up sitting there, unsure of what to do. It's an easy mistake to make when you need to modify many hosts and want to start the automation right away. The reason they ended up drawing a blank is that they weren't ready to effect changes on even a single host. They needed first to figure out how to reach the desired state.

This is the first rule of automation: *automation is simply a set of already working steps, tied together in an automated manner.*

This means that the first step toward automating a procedure usually involves manual changes! A development system (such as an SA's desktop UNIX/Linux system or a dedicated server system) is used to build, install, and configure software. You might need to perform these activities separately for all your site's operating systems and hardware platforms (SPARC vs. x86 vs. x86_64, etc.).

Here's an overview of the automated change development process:

- Make the change in a test environment.

- Make it fit your policy; for example, make it run as a nonroot user or install it in a specific directory tree.

- Automate the deployment steps.

- Test the deployment to a small number of testing or staging hosts and confirm that you achieve the desired effects.

- Deploy the change to all hosts using the newly developed automation.

So with automation, you simply take the solid work that you already do manually and speed it up. The side effect is that you also reduce the errors involved when deploying the change across all the systems at your site.

Exploring an Example Automation

In this section we'll take a set of manual steps frequently performed at most sites and turn it into an automated procedure. We'll use the example to illustrate the important points about creating automated procedures.

Scripting a Working Procedure

An SA needs to create user accounts regularly. In this case, you'll use several commands to create a directory on a central Network File System (NFS) server and send off a welcome e-mail. You must run the commands on the correct host because the accounts from that host are pushed out to the rest of the hosts.

To begin the automation process, the SA can simply take all the commands and put them into a shell script. The script might look as simple as this:

```
#!/bin/sh
useradd $1
cp /opt/admin/etc/skel/.* /home/$1/
```

Then the SA composes an e-mail to the new user with important information (having a template for the user e-mail is helpful). This procedure works, but another SA cannot use it easily. If it generates any errors, you might find it difficult to determine what failed. Plus, you might encounter problems because the script attempts all the steps regardless of any errors resulting from earlier steps. In just a few minutes, you can make some simple additions to turn this procedure into a tool that's usable by all SA staff:

```
#!/bin/sh
PATH=/sbin:/usr/sbin:/bin:/usr/bin
REQUIRED_HOST=adminhost1

usage() {
        echo "Usage: $0 account_name"
        echo "Make sure this is run on the host: $REQUIRED_HOST"
        exit 1
}
```

```
MYHOSTNAME=`hostname`

[ -n "$1" -a $MYHOSTNAME == $REQUIRED_HOST  ] || usage

USERNAME=$1

useradd -m $USERNAME || exit 1
cp /opt/admin/etc/skel/.bash* /home/$USERNAME/ || exit 1

/usr/bin/mailx -s "Welcome to our site" ${1}@example.net <<EOF
The SA team has created an account for you on the UNIX systems.
You have a default password that's unique to your account,
which will need to be changed upon initial login.
The system will force this password change.

Please call the SA help desk at 555-1212 in order to receive your
password, and to ask any questions that you may have.
EOF
```

Because the revised script ensures that it's running on the right host and that an argument is passed to it, it now helps the SA make sure it's not called incorrectly. This helps the author and any other users of the script. Having usage information should be considered mandatory for all administrative scripts, even if the scripts are meant to be used only by the original author.

Another advantage of scripting this procedure is that the same message is sent to all new users. Consistency is important for such communications, and it'll help ensure that new users are productive as soon as possible in their new environment.

Administrative scripts should not run if the arguments or input is not exactly correct. You could also improve the preceding script to ensure that the username supplied meets certain criteria.

Prototyping Before You Polish

The preceding script is still a prototype. If you were to give it an official version number, it would need to be something like 0.5, meaning that it's not yet intended for general release. Other SA staff members can run this functional prototype to see if it achieves the desired goal of creating a working user account.

Once this goal is achieved, the automation author can move on to the next step of polishing the script. The SA shouldn't spend much time on cosmetic issues such as more verbose usage messages before ensuring the procedure achieves the desired goal. Such things can wait.

Turning the Script into a Robust Automation

Now you want to turn the script into something you would consider version 1.0—something that will not cause errors when used in unintended ways. Every automation's primary focus should be to achieve one of two things:

- *A change to one or more systems that achieves a business goal*: The creation of a new user account falls into this category.

- *No change at all*: If something unexpected happens at any point in the automation, no changes should be made at all. This means that if an automated procedure makes several changes, a failure in a later stage should normally result in a rollback of the earlier changes (where appropriate or even possible).

   Your earlier user-creation script could use some improved error messages, as well as a rollback step. Give that a shot now:

```
#!/bin/sh
# Written by ncampi 05/26/08 for new UNIX user account creation
# WARNING!!!! If you attempt to run this for an existing username,
# it will probably delete that user and all their files!
# Think about adding logic to prevent this.

# set the path for safety and security
PATH=/usr/sbin:/bin:/usr/bin

# update me if we fail over or rebuild/rename the admin host
REQUIRED_HOST=adminhost1

usage() {
        echo "Usage: $0 account_name"
        echo "Make sure this is run on the host: $REQUIRED_HOST"
       exit 1
}

die() {
      echo ""
      echo "$*"
      echo ""
      echo "Attempting removal of user account and exiting now."
      userdel -rf $USERNAME
      exit 1
}
```

```
MYHOSTNAME=`hostname`

[ -n "$1" -a $MYHOSTNAME == $REQUIRED_HOST  ] || usage

USERNAME=$1

useradd -m $USERNAME || die "useradd command failed."
cp /opt/admin/etc/skel/.bash* /home/$USERNAME/ || \
die "Copy of skeleton files failed."

/usr/bin/mailx -s "Welcome to our site" ${1}@example.net <<EOF

The SA team has created an account for you on the UNIX systems.
You have a default password that's unique to your account,
which will need to be changed upon initial login. The system will
force this password change upon your first login.

Please visit the SA help desk in order to receive your password,
and to ask any questions that you may have.
EOF
```

It seems like a bad idea to trust that someone who calls your help desk claiming to be a new user is really the person in question, even if caller ID tells you the phone resides in your building. You might want to require that the user physically visit your help desk. If this isn't possible, the SA staff should come up with a suitable substitute such as calling the new user's official extension, or perhaps having the new user identify himself or herself with some private information such as a home address or employee number.

Attempting to Repair, Then Failing Noisily

The preceding script attempts a removal of the new user account when things go wrong. If the account was never created, that's okay because the `userdel` command will fail, and it should fail with a meaningful error message such as "No such account."

You'll encounter situations where a rollback is multistep, so you'll need to evaluate each step's exit code and indicate or contraindicate further rollback steps based on those exit codes. Be sure to emit messages about each step being taken and the results of those steps when the command is an interactive command. As the script author you know exactly what a failure means at each step, so be sure to relay that information to the SA running the script.

Each and every step in an automation or administrative script needs to ensure success; don't ever move on blindly with the assumption that a command worked. Even

something as simple as copying a few config files into a new user's home directory can fail when a disk fills up. Assumptions can and will bite you.

Focusing on Results

When in doubt, opt for simplicity. Don't attempt fancy logic and complicated commands when the goal is simple.

For example, you might have a script that takes a list of Domain Name System (DNS) servers and generates a `resolv.conf` file that's pushed to all hosts at your site. When a new DNS server is added or a server is replaced with another, you need to run the script to update the file on all your systems.

Instead of running the script to generate the file on each and every host at your site, you can run the command on one host, take the resulting output, and push that out as a file to all hosts. This technique is simple and reliable compared to the requirement of running a command successfully on every host. A complicated procedure becomes a simple file push. This is the KISS (Keep It Simple, Stupid) principle in all its glory. Our system administration experience has taught us that increased simplicity results in increased reliability.

CHAPTER 3

■ ■ ■

Using SSH to Automate System Administration Securely

The Secure Shell (SSH) protocol has revolutionized system administration ever since it became popular in the late 1990s. It facilitates secure, encrypted communication between untrusted hosts over an unsecure network. This entire chapter is devoted to SSH because it plays such an important part in *securely* automating system administration.

In this introductory chapter, we assume that you already have SSH installed and operating properly. We have based the examples in this book on OpenSSH 4.x using version 2 of the SSH protocol. If you are using another version of SSH, the principles are the same, but the implementation details might differ.

For a more thorough and complete discussion of SSH, we highly recommend *SSH, The Secure Shell: The Definitive Guide, Second Edition* by Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes (O'Reilly Media Inc., 2005).

SSH AND CFENGINE

The author of cfengine, Mark Burgess, has said that SSH and cfengine are "perfect partners." The SSH suite of programs provides secure communications for remote logins, and cfengine provides secure communications for system automation (along with the automation framework itself).

SSH and cfengine share the same distributed authentication model. SSH clients use a public-key exchange to verify the identity of an SSH server, with the option of trusting the remote host's identity the first time the host's key is seen. Cfengine also uses public-key authentication, although the cfengine server daemon also authenticates connecting clients for additional security. As with SSH, you can configure cfengine to trust the identity of other hosts upon initial connection.

We recommend you allow cfengine to trust the identity of other hosts in this manner. Doing so allows an SA to bring up a new cfengine infrastructure without the additional problem of key generation and distribution. If a host's keys change at any point in the future, cfengine will no longer trust its identity and will log errors.

Learning the Basics of Using SSH

If you are already familiar with the basic use of SSH, you might want to skim this section. If, on the other hand, you are an SSH novice, you are in for quite a surprise. You'll find that SSH is easy and efficient to use, and that it can help with a wide variety of tasks.

The commands in this section work fine without any setup (assuming you have the SSH daemon running on the remote host). If nothing has been configured, all of these commands use password authentication just like Telnet; except with SSH, the password (and all traffic) is sent over an encrypted connection.

Use this command to initiate a connection to any machine as any user and to start an interactive shell:

```
$ ssh user@host
```

You can also execute any command in lieu of starting an interactive shell. This code displays memory usage information on the remote host:

```
$ ssh user@host free
              total       used       free     shared    buffers     cached
Mem:         126644     122480       4164       1164      29904      36300
-/+ buffers/cache:       56276      70368
Swap:        514072      10556     503516
```

Finally, the scp command allows you to copy files between hosts using the SSH proto-col. The syntax resembles the standard cp command, but if a file name contains a colon, it is a remote file instead of a local file. As with the standard ssh command, if no username is specified on the command line, your current username is used. If no path is specified after the colon, the user's home directory is used as the source or destination directory. Here are a few examples:

```
$ scp local_file user@host:/tmp/remote_file
$ scp user@host:/tmp/remote_file local_file
$ scp user1@host1:file user2@host2:
```

The last example copies the file named file from user1's home directory on host1 *directly* into user2's home directory on host2. No file name is given in the second argu-ment, so the original file name is used (file, in this case).

Enhancing Security with SSH

Before SSH, the `telnet` command was widely used for interactive logins. Telnet works fine, except that the password (well, everything actually) is sent over the network in plain text. This isn't a problem within a secure network, but you rarely encounter secure networks in the real world. Machines on an unsecure network can capture account passwords by monitoring Telnet traffic.

IS YOUR NETWORK SECURE?

Some people define an unsecure network as the Internet and a secure network as anything else. Others think that as long as you have a firewall between a private network and the Internet that the private network is secure. The truly paranoid (such as ourselves) just assume that all networks are unsecure. It really depends on how much security you need. Are you a likely target for crackers? Do you store important, private information? Because nothing is ever 100 percent secure, we find it easier to assume networks are not secure and skip the rest of the questions.

  If you think you have a secure network, be sure to consider all the possible security vulnerabilities. Remember, employees within a company are often not as trustworthy or security-conscious as you would like. Somebody might have plugged in a wireless access point, for example. A person with more malicious intentions might deliberately tap into your private network, or exploit a misconfigured router or firewall. Even a fully switched network with strict routing can be vulnerable. We always try to be on the paranoid side because we'd rather be safe than sorry.

When it comes to automating system administration tasks across multiple systems, passwords are a real pain. If you want to delete a file on ten different machines, logging into each machine with a password and then deleting the file is not very efficient. In the past, many system administrators turned to `rsh` for a solution. Using a `.rhosts` file, `rsh` would allow a certain user (i.e., `root`) on a specific machine to log in as a particular user (again, often `root`) on another machine. Unfortunately, the entire authorization scheme relies on the IP address of the source machine, which can be spoofed, particularly on an unsecure network.

The most secure way to use SSH is to use password-protected public/private Rivest, Shamir, and Adleman (RSA) or Digital Signature Algorithm (DSA) key pairs. Access to any given account is granted only to users who not only possess the private key file, but also know the passphrase used to decrypt that file.

Another component of SSH is a program called `ssh-agent`. The program uses the passphrase to decrypt your private key, which is stored in memory for the duration of your session. This process eliminates the requirement that you enter the passphrase every time you need to use your private key.

Using Public-Key Authentication

Many SAs are more than happy to use SSH with its default password authentication. In this case, SSH simply functions as a more secure version of Telnet. The problem is that you need to enter a password manually for every operation. This can become quite tedious, or even impossible, when you are automating SA tasks. For most of the activities throughout this book, you must use RSA or DSA authentication.

Even if you use RSA authentication, you still need a passphrase to encrypt the private key. You can avoid entering the passphrase every time you use SSH in one of two ways. You can use an empty passphrase, or you can use the `ssh-agent` command as discussed in the next section. One major disadvantage of empty passphrases is that they are easy to guess, even by people with little skill.

SHOULD YOU USE AN EMPTY PASSPHRASE?

Some think that using an empty passphrase is one of the seven deadly sins of system administration. We think it can be appropriate within an isolated environment, especially when the security implications are minimal. For example, a Beowulf cluster generally has an internal private network containing only one machine with an external network connection. For instance, if a university uses the cluster for research, it might not be a target for intrusion. In this case, having an unencrypted private key on one of the cluster machines might not be too much of a concern.

However, if the same cluster were in use by a company doing important and confidential research, then, at the very least, the key should not reside on the one machine with an external connection. Of course, it would be even better to use an encrypted key along with `ssh-agent`. This key could be placed on a machine completely separate from the cluster, yet you could use it to access both the gateway and the individual nodes. This scenario would also remove the need to have the private-key file on the cluster at all, whether encrypted or not.

The most important thing to consider is what access the key provides. If the key provides `root` access to every system in your entire network, then the risks of leaving the key unencrypted (i.e., with no passphrase) are pretty great. But if the key allows the Dynamic Host Configuration Protocol (DHCP) server to be restarted on only one host, then what will an attacker do with it? Perpetually restart your DHCP server? Maybe—but that's not the end of the world, and it's easy to fix (change keys).

Version 2 of the SSH protocol supports two types of public-key encryption: RSA and DSA. The two encryption schemes are similar and generally considered to provide equivalent security. For no particular reason (apart from the fact that we are most familiar with it), we will use RSA for the examples within this book.

The main security difference in using RSA or DSA keys for login authentication is that the trust relationship changes. When you use password authentication, the server directly challenges the client. With public-key authentication, the challenge occurs at the client side. This means that if a user on the client side can get hold of a key, he or she will get into the system unchallenged. Thus the server has to trust the client user's integrity.

Generating the Key Pair

The first step in the key-generation process is to create your public- and private-key pair. OpenSSH provides a command just for this purpose. The following command creates a 2,048-bit RSA key pair and prompts you for a passphrase (which can be left blank if you so desire):

```
$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.
Your public key has been saved in ~/.ssh/id_rsa.pub.
The key fingerprint is:
3a:85:c7:e4:23:36:5c:09:64:08:78:b3:72:e0:dc:0d kirk@kaybee.org
```

The default output files are `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub` for the private and public keys, respectively.

WHAT SIZE KEY SHOULD YOU USE?

The bigger the key is, the harder it is to crack. Plus, a longer key length makes a key only slightly slower to use.

When choosing a key size, you must consider the value of the information or capabilities that the key protects. As long as your key would take more effort to crack than the data (or power) is worth, you are okay. An excessively large key places an unnecessarily large load on your systems.

If you are protecting data, you should also consider how long that data will be valuable. If the data will be worthless in one month and the key would take three months to crack, then the key is big enough. But be sure to consider that the attacker might have specialized hardware or advanced algorithms that can crack your key faster than you'd expect.

The size of the key makes the biggest speed difference during the actual key-generation process. Large keys are also more work (and therefore a little slower) when the computer encrypts and decrypts data. SSH uses RSA/DSA only when it initiates a new connection, so the key size affects only the initial session negotiations—not the performance of a session once it is established.

Throughout this book, we will generally show you examples that use the SSH key to access your systems. The actual data being sent is usually not important; it will typically contain commands to be executed and other control data. If somebody later decrypts this traffic, the results will probably be of little value.

But in some cases, the data being transferred *is* sensitive. In these instances, the RSA/DSA key is one of many things to consider because you use these protocols only to exchange keys for the algorithm used to encrypt the actual data. If attackers have captured the SSH session (i.e., using a network sniffer), they can crack the public key (by determining its associated private key) and determine the encryption key, or they can crack the actual encrypted data directly.

You can use the `-c` switch to the `ssh` command to control which cipher you use to encrypt your session. Your options with SSH protocol version 1 are `des`, `3des`, and `blowfish`—but you should avoid version 1 of the SSH protocol. With version 2, you have many bulk cipher options (including `blowfish`). Most SAs favor the `blowfish` cipher because it's fast and believed to be secure.

Specifying Authorized Keys

Now that you have a public-key file, you can simply place that key in any account on any machine running the SSH server (usually named `sshd`). Once you've set up the account properly, your private key will allow easy access to it. Determining the private key from a public key is virtually impossible, so only someone who has the private key can access the account.

To allow access to an account, simply create `~/.ssh/authorized_keys`. The file contains one key per line (although the lines are very long—the 2,048-bit RSA key created in the previous example is almost 400 characters long in its ASCII representation). If the file does not currently exist, you can simply make a copy of your public-key file.

You should also be careful with your permissions because `sshd` is usually very picky. In general, your home directory and the `~/.ssh` directory must be only writable by the user (and not their group, even if they have their own group). The directory must be owned by the user as well—this can be an issue if `root`'s home directory is `/` and it is not owned by `root`. If your RSA key is rejected, look in the logs on the system you are connecting to; they will usually tell you why.

Here is an example that assumes you have already copied your public-key file into your home directory in another account:

```
$ mkdir -p ~/.ssh
$ chmod 0700 ~/.ssh
$ cp ~/id_rsa.pub ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

To add a second key, simply append it to the file. Once you have the file in place, your private key alone allows you to access the account. Of course, by default, the account password also allows access to the account. You can disable this feature in the OpenSSH sshd by modifying `/etc/ssh/sshd_config` (or the equivalent on your system) and adding this line:

```
PasswordAuthentication no
```

Alternatively, you could completely disable the account password (usually stored in `/etc/shadow`) and allow only RSA-authenticated logins. However, this isn't a good idea if the user needs that password for other services such as POP3 mail access, FTP file transfers, and so on.

Using ssh-agent

If you can use `ssh-agent` to allow passwordless operation instead of leaving your private key unencrypted, then you will greatly add to the security of your systems. The `ssh-agent` program allows you to enter your passphrase only once per "session" and keeps your private key in memory, allowing passwordless connections for the rest of the session.

Knowing ssh-agent Basics

Using `ssh-agent` is simple. You start your command shell or your X session using the agent. Once logged in, you can run

```
$ ssh-agent bash
```

and you will have a new shell running through the agent. Or, if you use the wonderful screen program (included with most Linux installations and available from `http://directory.fsf.org/project/screen/`), you can use

```
$ ssh-agent screen
```

to begin your screen session. Use the following script as your `~/.Xclients` (or `~/.xinitrc`) to allow easy use of `ssh-agent` within X:

```
#!/bin/bash

cd ~
exec ssh-agent bin/startx-continue
```

As you can see, ssh-agent runs the startx-continue script. That script runs ssh-add </dev/null to add the key and prompt for a passphrase (/dev/null causes the program to use an X window for the passphrase entry). The startx-continue script also performs other startup tasks and finally starts the window manager.

These manual steps to start ssh-agent shouldn't be necessary on modern desktop environments; generally you'll already have an ssh-agent process running. To test, simply list the keys loaded into your agent:

```
$ ssh-add -l
```

If your output looks like this, you don't have an agent running and you should start one yourself as shown previously:

```
Could not open a connection to your authentication agent.
```

Once you are running the agent, you can add your private key(s) with ssh-add:

```
$ ssh-add
Enter passphrase for /home/kirk/.ssh/id_rsa:
Identity added: /home/kirk/.ssh/id_rsa (/home/kirk/.ssh/id_rsa)
```

When you use ssh-agent to run another command, that ssh-agent session exists for as long as that command runs (such as your X session). Once that command terminates, any stored keys are lost. This is fine when you can start your entire X session as you just saw, but what if you can't? You can use the ssh-agent command as shown in the next section to start a new ssh-agent for each login. This works well, unless you have a good number of simultaneous logins, in which case you will have to add your SSH keys for each session. If you are in this situation, consider using a tool called keychain that allows all your logins on the same system to share the same ssh-agent easily. You can find information about this tool at http://www-106.ibm.com/developerworks/library/l-keyc2/.

We generally recommend using screen. Whenever you spawn new shells inside screen, they'll each have the same environment, allowing you to use the same ssh-agent from each virtual screen. The additional benefits of screen are many, but we will mention only one here: you can log back in to a remote host and resume your session after an interruption arising from network or local computer problems. This benefit alone is worth a lot to an SA.

Getting Advanced with ssh-agent

You can also use ssh-agent without starting a new process. In the Bash shell (or any POSIX-compliant shell) you can, for example, start ssh-agent like this:

```
$ eval `ssh-agent`
```

Note the backticks around `ssh-agent`; they cause the output of this command to be passed to the `eval` command that will execute the code. In fact, all `ssh-agent` really does is start itself and print out some environment variables to be set by the shell. When you use `ssh-agent` to start a new process (as shown in the previous section), it simply sets these variables and creates a new process with the variables already set. You can run `ssh-agent` by itself to easily see what is set:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-XXoND8EO/agent.26962; export SSH_AUTH_SOCK;
SSH_AGENT_PID=26963; export SSH_AGENT_PID;
echo Agent pid 26963;
```

The `SSH_AUTH_SOCK` environment variable contains the path to the named socket that `ssh-agent` created to allow communication between the SSH program and the agent. The `SSH_AGENT_PID` variable contains the agent's process ID so that it can be killed at some point in the future.

The main disadvantage of running `ssh-agent` this way is that you must kill the agent through some external method if you want it to stop running once you have logged out. The more basic usage causes the agent to die upon completion of the process it executed.

Suppose you have a script that executes numerous SSH operations and you want to enter the passphrase only once. You could create the following script:

```
#!/bin/bash

# Start the agent (don't display PID)
eval `ssh-agent` >/dev/null
# Now, ask for the key once
ssh-add

# Now, perform a bunch of SSH operations
ssh host1 'command1'
ssh host1 'command2'
ssh host2 'command3'

# Finally, kill the agent and exit
kill $SSH_AGENT_PID
exit 0
```

This script would prompt you for the passphrase only once, store the private key in the agent, perform several operations, and then kill the agent when it was finished.

■**Note** You can find the code samples for this chapter in the Downloads section of the Apress web site (`http://www.apress.com`).

Forwarding Keys

You can configure your SSH client to *forward* your `ssh-agent` as well. If you enable this option, you can connect from machine to machine while your private key is in memory only on the original machine (the start of the chain). The key itself is never transmitted over the network. You'll find the agent useful when connecting to machines on private networks. You can connect to the gateway machine and then connect to internal machines that you cannot access directly from your workstation. For example, you can connect to one machine as `root` and run a script that connects to other machines using your private key, although your key does not actually exist on that machine.

BE CAREFUL WITH SSH-AGENT FORWARDING

You should never forward your `ssh-agent` connection to untrusted hosts (hosts where untrusted users have `root` access). The `root` users on other systems cannot obtain your actual private key, but they *can* use your forwarded `ssh-agent` connection to access other machines using *your* private key. OpenSSH lets you specify different options for different hosts (in `ssh_config`) so that you can forward your `ssh-agent` only to trusted hosts.

In addition, once you connect to another host and then use the `ssh` command on that host to connect to a third host, you are using the SSH client configuration of the second host, not the first host. That second host might have been configured to forward `ssh-agent` connections anywhere—including untrusted hosts.

So, prudent users forward their agents only to specific hosts. These select machines allow only trusted users access to the `root` account, and they also limit the hosts to which they will forward the `ssh-agent` session. You can also do this on the command line instead of modifying the actual `ssh_config` file; simply specify the option `-o "ForwardAgent no|yes"` to the `ssh` command.

Also note that, in the `authorized_keys` file, you can use the `from` directive to restrict which remote hosts are allowed to connect with the specified key (discussed next in the "Restricting RSA Authentication" section). If you forward your key only to certain systems, you can allow login only from those systems. If you accidentally forward your key to some other host, it won't work from that system anyway.

Some people also use `ssh-agent` in a noninteractive environment. For example, you might have a system-monitoring script that needs to connect to other machines continuously. You could manually start the script through `ssh-agent`, and then the script could run indefinitely using the passphrase you entered at startup. You could even place something like this in your system's startup scripts:

```
# start the ssh agent
/usr/bin/ssh-agent | /usr/bin/head -2 > ~/.ssh/agent-info

# alert oncall person to the system reboot
echo "$(hostname) rebooted, need to ssh-add the ssh keys into the ssh-agent" \
   | /bin/mail -s "$(hostname) rebooted" oncall@page.example.com
```

Any scripts that need access to this `ssh-agent` can source `~/.ssh/agent-info`. If attackers can access the system backups or steal the system disk, they'll gain the encrypted private-key file. But even though they'll have the private key, they won't be able to use it because they lack the passphrase to decrypt it. If you employed a passphrase-free private key instead, you'll need good backup security and physical security.

Restricting RSA Authentication

The `authorized_keys` file can contain some powerful options that limit the amount of account access the private key is granted. You can also use these options to prevent your agent from being forwarded to an untrusted host. To do so, place these options in the `authorized_keys` file at the beginning of the line and follow the entry with a space character. No spaces are allowed within the option string unless they are contained within double quotes. If you specify multiple options, you must separate them with commas. Here's a list of the options and a brief description of each (the `sshd` man page contains more detailed information):

`from="pattern-list"`: This option can specify a list of hosts from which the connection must be made. This way, even if the key (and the passphrase) is stolen, the connection still must be made from the appropriate host(s). The pattern could be `*.myoffice.com` to allow only hosts from the office to connect using that key.

`command="command"`: If specified, the given command always runs, regardless of what the SSH client attempts to run.

`environment="NAME=value"`: You can use this command—which you can list multiple times—to modify or set environment variables. The command is disabled by default for security reasons, but if you want its functionality you can enable it using the `PermitUserEnvironment` option in `sshd_config`.

`no-port-forwarding`: SSH allows ports on the server (or any machine accessible by the server) to be forwarded to the remote client. So, if users can connect to a gateway machine via SSH, they can forward ports from your private network to their remote machines, possibly bypassing some or all security. This prevents a specific key from forwarding any ports over its connection.

`no-X11-forwarding`: SSH can also forward X11 connections over the encrypted connection, allowing you (and the `root` user) to run X11 applications that display on the computer initiating the SSH connection. The `no-X11-forwarding` command disables this feature for the key in question.

`no-agent-forwarding`: This prevents an `ssh-agent` connection from being forwarded to the host when a user connects to it with the specified key.

`no-pty`: Prevents the allocation of a pseudo terminal so that an interactive login is not possible).

`permitopen="host:port"`: Allows only a given host and port to be forwarded to the remote client.

You can use these options for a lot of interesting tasks, as the following sections illustrate.

Dealing with Untrusted Hosts

When adding your public key to the `authorized_keys` file on an untrusted host, you could add some of the options just discussed to prevent agent and X11 forwarding. This is a good idea, but you shouldn't rely on it—if an untrusted `root` user on the machine can hijack your forwarded X11 or agent session, that user can probably also modify your `authorized_keys` file. That said, you can prevent the forwarding on both ends (client and server) to be extra safe. To do so, put the following in your `authorized_keys` file on the remote host (the key has been trimmed down for easier reading):

```
no-X11-forwarding,no-agent-forwarding,from="*.kaybee.org" ssh-rsa AB...YZ
```

This example also limits connections to this account. The key will be granted access only if the canonical hostname is `something.kaybee.org`.

Allowing Limited Command Execution

Suppose you have a script that monitors a set of servers. Root access is not necessary for monitoring the systems. The script does, however, need to reboot the machines in some

cases, which does require root access. The following configuration, when placed in `~root/`
`authorized_keys`, allows this specific key to reboot the system and nothing more:

```
no-port-forwarding,command="/sbin/reboot",no-pty ssh-rsa AB...YZ
```

Whoever possesses the specified private key cannot open an interactive shell or for-
ward ports. They can do only one thing: run the `/sbin/reboot` command. In this specific
example, you must be careful because if you connect to the account with the specified
key, the system will reboot (regardless of what command the remote client attempts to
run). You must also make sure you use an absolute path for the command. If you don't,
a malicious user might be able to place a command with the same name earlier in the
search path.

Forwarding a Port

Forwarding a port between two machines proves useful in many situations. If the port is
not encrypted, for example, you can use SSH to forward it over an encrypted channel. If
the machine is behind a firewall, that machine can connect to an outside machine and
forward ports to enable outside access.

Accessing a Server Behind NAT

Suppose you want to view a web page on a machine that resides on a private network but
can initiate outgoing connections using Network Address Translation (NAT). You can
connect from that web server to your desktop machine on another network using SSH:

```
$ ssh -R 8080:localhost:80 user@your-desktop-system
```

The command says to connect from the web server (which is behind the NAT router)
to the client (your desktop) and connect port 80 on the server to port 8080 on the client
(the desktop). Once this command has been executed, a user of the desktop system can
point a browser to port 8080 and view the content on port 80 of the web server.

You could replace the hostname `localhost` with the name of any other host that the
initiating host (the web server, in this example) can access. You can use this technique to
provide connectivity between two systems that could not normally communicate with
each other. Let's say, for example, that a router in the same private network as the web
server allows Telnet access through port 23. The web server could map port 23 on that
router to port 2323 on some other system:

```
$ ssh -R 2323:my-router:23 user@some-host
```

Once you run this command, you will actually have an interactive login session on
the destination system. As long as that session is open, the port forwarding is active.

Encrypting Mail Traffic

To forward unencrypted port 25 (mail) traffic from your client to a server over an encrypted channel, you could run this command as root on your local machine:

```
$ ssh -L 25:localhost:25 user@mailserver
```

(This doesn't work if a mail server is already running on the local machine because it is already using port 25.) When the command is executing, you could send mail to port 25 of your local machine and that traffic would really go to the mail server over the encrypted connection.

Configuring authorized_keys

If you want to create a special account on the mail server that allows users only to forward traffic to port 25, you could configure the authorized_keys file to restrict access to the account:

```
command="while true; do sleep 1000; done",no-pty,
 permitopen="localhost:25" ssh-rsa AB...YZ
```

Please note that the preceding code would be only one line in the actual authorized_keys file, with no space after the no-pty,. This configuration allows you to make a connection that runs an infinite loop and forwards port 25—that's all. When connecting with this specific key, you cannot do anything else with this account.

Using SSH for Common Accounts

One interesting way to use SSH involves allowing several users to access one or more common accounts. You'll probably find this practice most useful for the root account (when there are multiple administrators), but you could also use it for other accounts (such as a special account to do software builds). The advantage of this approach is that each user does not have to know the account password to access the account. In addition, the logs can tell you who is actually logging into the account.

Another, and perhaps better, solution is to have each user log in with his or her user account. The user can then use sudo to execute certain commands as root (we introduced sudo in Chapter 1). But sudo is not always an option—particularly if you don't want to create a user account on the system for each user who needs to run commands as root.

Preparing for Common Accounts

The setup is simple. You generate a key pair for each user and then list the appropriate
public keys in the account's authorized_keys file. However, you might find it frustrating to
maintain this system manually when you have a large number of accounts and/or users.
It is much easier to create a configuration file:

```
# The account name is given first, followed by a colon,
# with each user who should be able to access that account
# listed afterward, and separated by commas.
root:amber,bob,frank,jill
build:amber,bob,susan
```

Then create a script that can process the configuration file and generate all the
authorized_keys files. This particular script assumes that each person's public key is in his
or her home directory and that he or she is using RSA:

```
#!/usr/bin/perl -w
use strict;

# Set the location of the configuration file here
my $config = "/usr/local/etc/ssh/accounts";

# Where the key fingerprints will be stored
# (for purposes of log analysis)
my $prints = "/usr/local/etc/ssh/prints";

# Set the path to a user's public key relative to
# their home directory
my $public_key = ".ssh/id_rsa.pub";

# This function takes one scalar parameter (hence the $
# within the parenthesis). The parameter is stored in
# the local variable $username. The home directory
# is returned, or undef is returned if the user does
# not exist.
```

```perl
sub GetHomeDir ($) {
   my ($username) = @_;
   my $homedir = (getpwnam($username))[7];
   unless ($homedir) {
      print STDERR "Account $username doesn't exist!\n";
   }
   return $homedir;
}

# This function takes in an account and the home directory and logs
# the key fingerprint (by running ssh-keygen -l), which has output:
# 2048 85:2c:6e:cb:f6:e1:39:66:99:15:b1:20:9e:4a:00:bc ...
sub StorePrint ($$) {
   my ($account, $homedir) = @_;
   my $print = `ssh-keygen -l -f $homedir/$public_key`;
   # Remove the carriage return
   chomp($print);
   # Keep the fingerprint only
   $print =~ s/^\d+ ([0-9a-f:]+) .*$/$1/;
   print PRINTS "$account $print\n";
}

# This function takes one line from the config file and
# sets up that specific account.
sub ProcessLine ($) {
   my ($line) = @_;
   # A colon separates the account name and the users with access
   my ($account, $users) = split (/:/, $line);
   my $homedir = GetHomeDir($account);
   return unless ($homedir);

   print "Account $account: ";

   # First, make sure the directory exists, is owned
   # by root, and is only accessible by root
   my $group = 0;
```

```perl
    if (-d "$homedir/.ssh") {
        $group = (stat("$homedir/.ssh"))[5];
        system("chown root:root $homedir/.ssh");
        system("chmod 0700 $homedir/.ssh");
    } else {
        mkdir("$homedir/.ssh", 0700);
    }

    # Remove the existing file
    unlink ("$homedir/.ssh/authorized_keys");

    # Create the new file by appending other users' public keys
    my ($user, $homedir2);
    foreach $user (split /,/, $users) {
        # Get this other user's home directory too
        $homedir2 = GetHomeDir($user);
        next unless ($homedir2);

        if ((not -f "$homedir2/$public_key") or
            (   -l "$homedir2/$public_key") ) {
          print "\nUser $user public key not found or not a file!\n";
          next;
        }

        print "$user ";
        my $outfile = "$homedir/.ssh/authorized_keys";
        system("cat $homedir2/$public_key >> $outfile");
        StorePrint($user, $homedir2);
    }
    print "\n";

    # Now, fix the permissions to their proper values
    system("chmod 0600 $homedir/.ssh/authorized_keys");
    system("chown $account $homedir/.ssh/authorized_keys");
    system("chown $account $homedir/.ssh");
    if ($group) {
        # We saved its previous group ownership... restore it.
        system("chgrp $group $homedir/.ssh");
    }
}
```

```perl
# Open the fingerprint file
open (PRINTS, ">$prints") or die "Can't create $prints: $!\n";

# Open the config file and process each non-empty line
open (CONF, "$config") or die "Can't open $config: $!\n";
my $line;
# The angle operators (<>) read one line at a time
while ($line = <CONF>) {
    chomp($line);
    # Remove any comments
    $line =~ s/\#.*$//;
    # Remove leading and trailing whitespace
    $line =~ s/^\s+//;
    $line =~ s/\s+$//;
    # Process the line (if anything is left)
    $line and ProcessLine($line);
}
close (CONF);
close (PRINTS);
exit 0;
```

ALWAYS WATCH FOR RACE CONDITIONS

You might find it odd that the `authorized_keys` file-generation script changes ownership of the `.ssh` directory to user `root` and group `root` and then changes it back to the proper user later in the script. The script makes these ownership changes to prevent any race-condition exploits by the user of that account. Even if you trust all your users now, you might not trust them all in the future, so you're better off addressing the problems while you write the original script.

The script first makes sure the directory is owned by `root` and writable by nobody else. Then it removes the current `authorized_keys` file. If this is not done, the current `authorized_keys` file could be a symbolic link to a system file that is overwritten when you create the file.

The script also checks the user's public-key file to make sure it is a regular file (the `-f` operator) and not a symbolic link (the `-l` operator). If the user's public-key file is a symbolic link, the account's user could point that link to any system file he or she could not normally read (such as the shadow password file). Then the script, when run, would copy the contents of that file into an `authorized_keys` file.

Note that you must remove the current `authorized_keys` file and check the public-key file after the `.ssh` directory's ownership and permissions change. If you do not, the user could theoretically change the files after you have checked them but before you access them, effectively bypassing all the security in the script.

As you can see, the script assumes all the home directories are on this particular machine. You can use various methods to synchronize home directories among multiple machines, as discussed in Chapter 7 and elsewhere throughout the book. Alternatively, you could easily modify this script to manage accounts on other machines using scp to transfer the actual authorized_keys files. Here's the output from this script when it is run with the sample configuration file:

```
$ ./account-ssh-setup.pl
Account root: amber bob frank jill
Account build: amber bob susan
```

The script also creates a file that lists all the key fingerprints and their associated account names. Later, you can use this file to aid in the analysis of the sshd log entries. The file, as you will notice, might contain duplicate entries, but that won't affect how it's used later.

Monitoring the Common Accounts

If you want to monitor which users are logging into which accounts, you must first keep a log of which key logs into which account. Unfortunately, OpenSSH does not do this by default. You need to turn up the logging level of sshd by adding this line to /etc/ssh/sshd_config (or wherever it is on your system):

```
LogLevel VERBOSE
```

Once you have added this configuration line and restarted sshd, you will see these logs (in /var/log/secure or wherever you have your other sshd logs). We've removed the headers for easier reading:

```
Found matching RSA key: cc:53:13:85:e5:a0:96:c9:24:f5:de:e0:e3:9e:9b:b6
Accepted publickey for test1 from 10.1.1.1 port 55764 ssh2
```

Unfortunately, the information you need for each login spans two lines in the log file, which makes analysis slightly more complicated. Here is an example script that can analyze a log file and summarize user logins (as with every example in this book, this script is only an example; you should modify it as necessary for your specific needs):

```perl
#!/usr/bin/perl -w
use strict;

# The logfile to analyze by default on a RedHat-based system
my $log = "/var/log/secure";
```

```perl
# Where the key fingerprints are stored
my $prints = "/usr/local/etc/ssh/prints";

# First, read and store the fingerprints in a hash table
# Duplicate lines will not hurt anything
open (PRINTS, "$prints") or die "Can't open $prints: $!\n";
my (%Prints, $line);
while ($line = <PRINTS) {
   chomp($line);
   my ($account, $print) = split / /, $line;
   $Prints{$print} = $account;
}
close (PRINTS);

# Open the logfile and process each line
# Store results in a two-tier hash table
open (LOG, "$log") or die "Can't open $log: $!\n";
my (%Results, $user);
while ($line = <LOG) {
   chomp ($line);
   if ($line =~ /Found matching \S+ key: ([0-9a-f:]+)/) {
      # Determine user from print-lookup hash (if possible)
      if ($Prints{$1}) {
         $user = $Prints{$1};
      } else {
         $user = 'Unknown';
      }
   } elsif ($line =~ /Accepted publickey for (\S+)/) {
      $Results{$1}{$user}++;
   }
}
close (LOG);

# Display the results
my $account;
foreach $account (keys %Results) {
   print "$account:\n";
   foreach $user (keys %{$Results{$account}}) {
      print "   $user: $Results{$account}{$user} connection(s)\n";
   }
}
exit 0;
```

Here's an example of the script being executed:

```
$ ./sshreport.pl
root:
   amber: 2 connection(s)
   bob: 1 connection(s)
build:
   susan: 4 connection(s)
```

The script is fairly simple, but you could expand it to support date ranges or to report the dates of the various logins.

■ ■ ■

Configuring Systems with cfengine

So far we've been discussing automation in a general way. At this point we'll move beyond single ad hoc measures to a more systematic and robust approach. While you certainly have the option of writing your own collection of automation scripts, we recommend you use a proven automation framework: cfengine.

Getting an Overview of cfengine

Cfengine is software you can use to automate changes on UNIX (and UNIX-like) systems. It is a high-level language that describes system state, not a general-purpose programming language such as Perl or a shell. It's primarily declarative, meaning that the SA writes out a technical description instead of a list of low-level steps to accomplish the goal. It is high-level enough that someone familiar with UNIX concepts and usage can read a cfengine configuration and understand what is being done without any prior cfengine knowledge.

The language drives what you should consider your personal software robot. This robot (called `cfagent`) does your repetitive work for you while you move on to other tasks.

In this chapter we'll use the current version of cfengine at the time of this writing: version 2.2.7.

Defining cfengine Concepts

Cfengine was designed to save time and reduce errors though automation. Its second, related goal is to enable computer systems to self-correct errors. It might take you some time to set up and configure cfengine, but you will be happier when everything has been said and done.

At first, performing a new task with cfengine might take longer than performing the same task manually. But when you upgrade the operating system and lose a change made under the old OS installation, you'll be glad you used cfengine because it will simply perform the change again. Or, when you realize a few other systems need the same change,

you can use cfengine to make this happen in seconds (by adding the new systems to the appropriate class).

If you made the change manually, on the other hand, it might take some time before you even notice that the change was lost. Once you notice, you'll have to make the change manually all over again—that is, of course, if you remember how you did it the last time. If ten new systems need a specific change, you might spend an hour changing each system yourself, whereas cfengine could have just done it for you.

Cfengine allows you to use one set of configuration files. Each host can transfer the configuration files from one or more cfengine servers before each run. As long as you make all the changes in that set of configuration files, all systems will receive the configuration updates automatically. You will no longer need to remember to make manual system changes. You will no longer need to use special scripts for special systems and/or scripts that have so many conditionals (based on hostname, operating system, etc.) that they've become unreadable and difficult to maintain. Cfengine comes with a rich set of automatically detected UNIX characteristics that the SA can use to perform tasks on only the desired systems.

Perhaps most important, this set of configuration files documents every change you make to every system. If you put a few comments in the files along with the commands, you will document not only what you have done but also *why* you did it.

CENTRALIZED CFENGINE CONFIGURATION FILES

Cfengine doesn't force centralized configuration files onto its users. In our examples, we choose to maintain a single configuration-file tree and distribute it to all hosts, and cfengine allows us to update configuration files any way we choose. Some sites choose to retrieve some or all files directly from a revision-control system such as Concurrent Versions System (CVS) or Subversion on all client systems. Some sites have configuration files copied from multiple remote servers to create a single configuration tree, in what would be considered a decentralized model.

For some tasks, cfengine abstracts the desired action from the technical specifics of the underlying operating system. For other tasks (namely editing files), cfengine provides an editing-specific command that allows you to specify the modifications exactly. Using these commands is similar to using `sed`. The cfengine text-editing commands have low-level abilities in addition to higher-level ones. We will cover the `editfiles` feature of cfengine later in this chapter.

Cfengine doesn't provide native support for certain tasks, but it lets you execute external scripts based on a system's class membership. When possible, you should use the internal commands that cfengine provides. If you don't, you can use custom shell and Perl scripts, but you should still get cfengine to execute them on your behalf.

Once you decide to use cfengine, you'll want to use it for as many tasks as possible. So you'll probably need to change your habits because you might be tempted to just "fix it real quick" instead of going through the proper cfengine process. The quotes around the word "quick" are carefully positioned. If you do a manual "quick fix" to your existing set of systems, a newly deployed system might be missing the change. When you work to redeploy the change using cfengine, you'll have to figure out how the change was made, test it, and deploy it once again. The simple act of figuring out how to make the change again is time-consuming. Using cfengine to deploy the change in the first place ensures you don't have to go through that process again for your existing systems and configuration, at least until you upgrade your OS or major applications.

You don't have to use cfengine to control all aspects of system configuration, so you can easily introduce it into an existing management framework without eliminating any existing methods of system configuration. You are free to use cfengine to control only the aspects of your systems that you're initially comfortable with. Over time, you can migrate the old configuration methods into cfengine. This situation isn't ideal for the long term because having two administration frameworks incurs increased complexity, but it will help you get comfortable with cfengine before committing all your site's administration to it.

Once you switch to cfengine, you will enjoy many benefits:

- *A standardized configuration for all hosts on your network that you can use to enforce homogeneity or to support diversity, each in an automated manner*: Cfengine configuration rules are each essentially a promise about the nature of the system. The `cfagent` program ensures that promises are kept.

- *The ability to change specific systems*: You can classify systems using a variety of built-in methods and classes (even ones added by the SA staff), and make changes only on the appropriate systems.

- *The ability to record system changes and perform them again if necessary*: One of cfengine's primary goals is to bring systems into conformance with policy and *keep* them that way.

- *Systems that might have intermittent uptime or network connectivity but will eventually make any necessary changes*: You won't need to keep track of what systems were up when you made a particular change across all your systems.

Evaluating Push vs. Pull

Yet another advantage to using cfengine is that it pulls from a server instead of pushing from the master system. This doesn't make a big difference when you have a local set of reliable servers that are usually up and running. But the pull method is much

more reliable if your systems are spread out over an unreliable network or if they aren't always running.

For example, if some systems can boot to either Linux or Windows, they can pull from the server whenever they are running Linux. If you were to use a push technique instead, the system might get neglected if it's running Windows at the time of the attempted push because the master push system expects a Linux host to be running at the remote IP address. When such systems boot into Linux again, cfengine will copy the latest configuration files to the system and ensure that the current promises are kept.

Another problem situation arises when you have UNIX running on one or more laptops that are not always connected to the network. A system like this might never be updated using the push method, because it would have to be connected to the network at the exact time a push occurs. With the pull method, the laptop would automatically pull the configuration changes the next time it contacts the configuration server.

Pull also scales more manageably because each host can decide where to get its updates and can fail over to an alternative if the request times out or otherwise fails. In a push model where multiple central hosts attempt to push to the same client at once, conflicts are likely to occur. Cfengine's author Mark Burgess says: "Push methods are basically indistinguishable from an attack on the system. They attempt to remove each user's local right to decide about their own systems. In a world where we increasingly own the devices we use as personal objects, this all seems a bit like something from the cold war." Mark is speaking to the freedom that both the pull model and cfengine allow. In this book we set standardized schedules and policies on our systems, but this is strictly a local policy choice. The cfengine framework is remarkably free of assumptions and requirements, and you can use it to implement the appropriate policies for your site.

Although cfengine typically pulls from a server and executes at regular intervals (configurable, but defaults to once an hour), it also supports the ability to force updates to all or any subset of the systems on demand. Obviously, you will find this useful when you are performing mission-critical bug fixes (e.g., something else you did messed up a system or two and you need to fix it very quickly).

You can also run cfengine directly on each system by logging in and manually running the `cfagent` command. Cfengine follows a good theory in system-administration automation: the more ways you can initiate changes to a system, the better—as long as all changes are done in the same way. In other words, cfengine provides several methods for updating each system, but all of them use the same configuration files and operate exactly the same way (once initiated). We can thank `cfagent` for this standardization; it's always the program reading the configuration files and implementing the policy.

Delving into the Components of cfengine

The cfengine suite consists of several compiled programs. Modern systems have plenty of disk space to house the binaries locally. Some of the older cfengine documentation implied that it's wise to share out `/usr/local` or a similar directory through NFS—or Andrew File System (AFS) or Distributed File System (DFS)—and utilize the shared files from all systems. But most SAs today would see a real disadvantage to the single point of failure inherent in the remote mounting of critical software like cfengine, at least when disk space is as cheap as it is.

We'll list some of cfengine's programs here:

`cfagent`: The autonomous configuration agent (the heart of the framework). This command can be run manually (on demand), by `cfexecd` on a regular basis, and/or by `cfservd` when triggered by a remote `cfrun` invocation. The necessary and sufficient condition for using cfengine is to run `cfagent` somehow.

`cfservd`: The file-transfer and remote-activation daemon. You must run this on any cfengine file servers and on any system where you would like to execute `cfagent` remotely.

`cfexecd`: The execution and reporting daemon. You run this either as a daemon or as a regular cron job. In either case, it handles running `cfagent` and reporting its output.

`cfkey`: Generates public/private key pairs and needs to be run only once on every host.

`cfrun`: You can run this command from a remote system that will contact the clients (through `cfservd`) and tell them to execute `cfagent`.

For any given command, you can see a summary of its options by using the `-h` command-line option. When running a command, you can always specify the `-v` switch to see detail. For example, `cfagent` is nonverbose by default but will describe the actions it takes when the `-v` switch is used.

When debugging a program, you should use the `-d2` switch to view debugging information (and, for daemons, the `-d2` switch prevents them from detaching from the terminal).

Mapping the cfengine Directory Structure

You must install the binaries in a directory mounted on every host or install them independently on each host. Everything cfengine uses during its normal operation is located under the `/var/cfengine/` directory by default, although Debian and its derivative distributions use `/var/lib/cfengine2` by default. The directory's contents are as follows:

bin/: Important binaries (cfagent, cfexecd, and cfservd) are usually copied here to ensure they are available when needed. Normal operation doesn't require this, but the cfengine example documentation recommends it and many sites adhere to it.

inputs/: This is the standard location for all the configuration files cfengine needs. We will initially use three files from this directory—cfagent.conf, cfservd.conf, and update.conf—but we'll be expanding on those initial files quite a bit as the book progresses.

outputs/: This is where the output from each cfexecd run is logged.

ppkeys/: This is where this system's public and private keys, as well as other systems' public keys, are located.

Managing cfengine Configuration Files

Each system must have a minimal number of configuration files. You should place these in the /var/cfengine/inputs/ directory on each system (/var/lib/cfengine2/inputs on Debian and derivatives), but you can maintain them in a central location and copy them (using a pull) to all client systems:

update.conf: This file *must* be kept simple. It is always parsed and executed by cfagent first. Its main job and intended usage is to copy the set of configuration files from the server. If any of the other configuration files contains an error, this file should still be able to update files so that the next run will succeed. If this file contains an error, you will have to fix any affected systems manually. At most sites this file should go unchanged for long periods of time once the cfengine infrastructure is initially set up.

cfagent.conf: This file contains the guts of your automation system—all the actions, group declarations, and so on. Throughout this book we utilize the cfagent.conf file as the starting point for all the included files (using cfengine's import feature) that make up the bulk of the configuration. This approach leads to more modular and easy-to-read configurations.

cfservd.conf: As the name suggests, this is the configuration file for the cfservd daemon. It defines which hosts can remotely execute cfagent and which remote hosts can transfer particular files.

You should manage the master copy of the configuration files with a source-control system such as Subversion. This way you have a record of any changes made and you can revert to an older configuration file if you introduce a problem into the system. If you feel that you don't have the time to set up Subversion or even CVS, then something extremely simple such as Revision Control System (RCS) will still be better than nothing. We'll

give examples on how to set up and use Subversion with cfengine in Chapter 11 (using cfengine itself to fully automate the process).

WHERE IS MY /VAR/CFENGINE/INPUTS DIRECTORY?

The `/var/cfengine/inputs` and `/var/cfengine` directories are not included with UNIX or Linux default installations. They might not even exist on systems with cfengine installed. Package-based cfengine installations often create some of the required cfengine directories, but usually not all.

It is up to you to create and configure the required directories and configuration files needed for your site's cfengine framework. We'll describe a simple cfengine site configuration later in this chapter, but the example is intended for demonstration purposes only. In Chapter 5, we'll configure a complete cfengine framework and build on it in later chapters.

For now, simply be aware that by default `cfagent` (or any other cfengine program) doesn't automatically configure a system in a way that satisfies all the prerequisites for running cfengine. You'll have to handle all the details yourself. If you're chomping at the bit to see how it's done, check out the `cf.preconf` preconfiguration script from Chapter 5.

Identifying Systems with Classes

The concept of classes is at the heart of cfengine. Each system belongs to one or more classes. Or, to put it another way: each time `cfagent` runs, many classes are defined based on a variety of different kinds of information. Each action in the configuration file can be limited only to certain classes. So, any given action could be performed on one host only, on hosts running a specific operating system, or on every host. Cfengine uses both built-in and custom classes, both of which will be discussed within this section.

Categorizing Predefined Classes

The host itself determines many of the classes that are defined—its architecture, hostname, IP address(es), and operating system. Several classes are also defined based on the current date and time.

To determine which standard classes are defined on any given system, run this command:

```
# cfagent -p -v | grep Defined
Defined Classes = ( 192_168_1 192_168_1_1 64_bit Day25 Debian_GNU_Linux_4_0__n__l_
Hr00 Hr00_Q4 June Min55_00 Min57 Q4 VMware Wednesday Yr2008 addr_ any campin_net
cfengine_2 cfengine_2_1 cfengine_2_1_20 compiled_on_linux_gnu debian debian_4
```

```
debian_4_0 fe80__290_27ff_fee8_9510 fe80__290_27ff_fee8_9511 ipv4_192 ipv4_192_168
ipv4_192_168_1 ipv4_192_168_1_1 linux linux_2_6_18_5_amd64 linux_x86_64
linux_x86_64_2_6_18_5_amd64
linux_x86_64_2_6_18_5_amd64__1_SMP_Sat_Dec_22_20_43_59_UTC_2007 net
net_iface_eth0 net_iface_eth1 net_iface_lo net_iface_ppp0
net_iface_ppp1 foo foo_campin_net foo_int x86_64 )
```

As you can see, this example system contains quite a number of predefined classes. They fall into a few categories:

Operating System: debian debian_4 debian_4_0

Kernel: linux linux_2_6_18_5_amd64

Architecture: x86_64 linux_x86_64 linux_2_6_18_5_amd64

Hostname: campin_net foo foo_campin_net

IP Address: ipv4_192 ipv4_192_168 192_168_1 192_168_1_1

Date/Time: Wednesday Yr2008 Hr00 Hr00_Q4 June Min55_00 Min57 Day25

Every system is a member of the any class, for obvious reasons. As you can see, cfengine provides good granularity with its default class assignments. We cannot possibly list all the classes that could be assigned to your systems, so you will have to check the list on each of your systems (or, at least, each type of system on your network).

The time-related classes probably require some additional explanation. The Min55_00 class specifies the current five-minute range. The Q4 class is always set in the last quarter of the hour. The Hr00_Q4 class says you are currently in the last quarter of the midnight hour (it's time for bed).

Defining Custom Classes

Custom classes are defined in the classes section of the cfagent.conf configuration file. Here is an example:

```
classes:
  # Check to see if X11R6 is installed
  X11R6                                 = ( '/usr/bin/test -d /usr/X11R6' )

  # Use built-in directory check for a local program directory
  have_usr_pkg_ganglia_3_0_7       = ( IsDir(/usr/pkg/ganglia-3.0.7) )
```

```
# Mail servers must be explicitly defined at our site
mail = ( mail1 mail2 )

# DNS and web servers are obvious by their configuration files
dns = ( '/usr/bin/test -f /etc/named.conf' )
web = ( '/usr/bin/test -f /etc/httpd/conf/httpd.conf' )

# We have a large number of web servers, let's break them out into
# groups of ten for when we implement changes across them
first_ten_webs      = ( RegCmp("webserver[0-9]i","${host}") )
second_ten_webs   = ( RegCmp("webserver1[0-9]i","${host}") )
third_ten_webs        = ( RegCmp("webserver2[0-9]i","${host}") )
fourth_ten_webs    = ( RegCmp("webserver3[0-9]i","${host}") )
fifth_ten_webs        = ( RegCmp("webserver4[0-9]i","${host}") )
# ...and so on...

# Any critical servers are a member of this class
critical = ( mail dns web )
```

When a class definition contains a quoted string, that is a command to be executed. If the command returns an exit code of 0, then this system will be part of that class.

Class definitions can (and often do) list other classes. If a system is a member of any of the listed classes, then it is also a member of the new class.

Some cfengine commands can define new classes in certain situations. If, for example, a particular drive is too full, a command can define a class and print a warning. Other parts of the configuration file might take further action if that class is defined. We'll explore this quite a bit in later chapters.

Finding More Information About Cfengine

You can download cfengine from its web site: http://www.cfengine.org/. The web site includes a tutorial, a comprehensive reference manual, mailing lists, and archives. Two additional web sites are useful. The first, http://www.cfengine.com, is a commercial venture started by the cfengine author and some colleagues. It contains enhanced documentation in return for a little information about how you use cfengine. The second is http://www.cfwiki.org, a community-run site with a lot of useful tips and tricks for dealing with cfengine, generally from very experienced cfengine users.

You should also examine the large number of sample configuration files included with the cfengine distribution.

Learning the Basic Setup

Within this section, we'll illustrate and discuss a simple cfengine setup that will provide a good framework for customization and expansion. These simple configuration files will not make many changes to your systems, but they will still show some of the power of cfengine.

This simple setup includes one central server and one other host. With cfengine, all hosts are set up identically (even with only slight differences on the server), so you could extend this example to any number of systems. We would recommend, though, that you start out with just two systems. Once you get cfengine up and running on those systems, expanding the system to other hosts is easy enough. In later chapters we'll completely overhaul this basic configuration to scale up to complete site-wide management.

Setting Up the Network

Before starting with cfengine, you should make sure your network is properly prepared. Using cfengine with dynamic IP addresses is difficult because cfengine utilizes two-way authentication during network communications. Even if you use the Dynamic Host Configuration Protocol (DHCP) to assign addresses to some or all of your systems, it should always assign the same IP address to systems that you'll control with cfengine. In other words, it doesn't matter which method you use to assign the IP addresses, as long as the IP address for each system to be managed stays consistent. Cfengine has configuration directives allowing it to understand that hosts on certain IP ranges use dynamic IP addresses, but this defeats the two-way trust mechanism. You should avoid dynamic IP addresses if possible.

The next task is to make sure your Domain Name System (DNS) is properly con-figured for your hosts. Each host should have a hostname, and a DNS lookup of that hostname should return that host's IP address. In addition, if that IP address is looked up in DNS, the same hostname should be returned.

If this setup is not possible, we recommend that you add every host to the `/etc/hosts` file on every system, although if your DNS isn't properly configured you'll have pain in other areas. If you are using a multihomed host, you must pay attention to which IP address will be used when your host is communicating with other cfengine hosts.

Running Necessary Processes

In the simplest setup, you can use cfengine by running `cfagent` on each system manually. You will, however, benefit more from running one or two daemons on each system.

The cfexecd Daemon

Although you could, theoretically, run cfagent only on demand, you're better off run-
ning it automatically on a regular basis. This is when cfexecd comes in handy; it runs as
a daemon and executes cfagent on a regular, predefined schedule. You can modify this
schedule by adding time classes to the schedule setting in the control block in cfagent.
conf. The default setting is Min00_05, which means cfagent will run in the first five minutes
of every hour. To run twice per hour, for example, you could place the following line in
the control section of cfagent.conf:

```
schedule = ( Min00_05 Min30_35 )
```

The cfexecd daemon does not have its own configuration file; it uses settings
intended for cfexecd in the cfagent.conf file.

You can also run cfexecd on a regular basis using the system's cron daemon. You
could add the following entry to the system crontab (usually /etc/crontab) to execute
(and report) cfagent every hour:

```
0 * * * * root /usr/local/sbin/cfexecd -F
```

The -F switch tells the cfexecd program not to go into daemon mode because it is
being run by cron.

For increased reliability, you can run cfexecd as a daemon and also run it from cron
(maybe once per day). You can then, in cfagent.conf, check for the crontab entry and
check whether the cfexecd daemon is running. The following lines, if placed in cfagent.
conf, perform these checks and correct any problems:

```
editfiles:
   { /etc/crontab
      AppendIfNoSuchLine "0 * * * * root /var/cfengine/bin/cfexecd -F"
   }

processes:
   "cfexecd" restart "/var/cfengine/bin/cfexecd"
```

With this technique, if either of the methods is not working properly, the other
method ultimately repairs the problem.

The cfservd Daemon

The cfservd daemon is not required on all systems. It needs to run on any cfengine file
servers, which, in our case, is the central configuration server only. It also allows you
to execute cfagent from other systems remotely. If you want this functionality (which

we recommend), then `cfservd` needs to be running on every system. In either case, you should always check whether it's running by using the following entry in `cfagent.conf`:

```
processes:
  "cfservd" restart "/var/cfengine/bin/cfservd"
```

Running `cfservd` on all hosts presents little risk, and it allows added flexibility when you later need to retrieve information from client systems using cfengine. Make sure that your access controls (the `admit` lines in `cfservd.conf`) don't allow access to unnecessary hosts. Only access explicitly defined in `cfservd.conf` is allowed, so `cfservd` is safe by default.

Creating Basic Configuration Files

You need to place your configuration files in the master configuration directory on the configuration server (as we'll explain in the next section). These common files will be used in their exact original form on every server in your network.

Example cfservd.conf

This is the configuration file for the `cfservd` daemon. It allows clients to transfer the master set of configuration files and also allows you to execute `cfagent` remotely using `cfrun`. Obviously only one system needs to allow access to the central configuration files (the server), but having `cfservd` allow access to those files doesn't hurt anything on other systems (because those systems don't have the files there to copy). All systems, however, can benefit from allowing remote `cfagent` execution, because it allows you to execute `cfagent` on demand from remote systems.

So, you can use the following `cfservd.conf` on all your systems:

```
control:
  domain                     = ( mydomain.com )
  AllowUsers                 = ( root )
  cfrunCommand               = ( "/var/cfagent/bin/cfagent" )
  TrustKeysFrom              = ( 10.1.1 )
  AllowConnectionsFrom       = ( 10.1.1 )
  AllowMultipleConnectionsFrom = ( 10.1.1 )

admit:
  /usr/local/var/cfengine/inputs *.mydomain.com
  /var/cfagent/bin/cfagent       *.mydomain.com
```

The cfrunCommand setting specifies the location of the cfagent binary to be run when a connection from cfrun is received. The admit section is important because it specifies which hosts have access to which files. You must grant access to the central configuration directory and the cfagent binary. You also need to grant access to any other files that clients need to transfer from this server.

Basic update.conf

You should keep the update.conf file as simple as possible and change it rarely, if ever. The cfagent command parses and executes the update.conf file before it does the same to cfagent.conf. If you put out a bad cfagent.conf, the next time the clients execute cfagent they get the new version because their update.conf file is still valid. Distributing a bad update.conf would be a bad idea, so we recommend testing any changes thoroughly before you place the file in the central configuration directory. We also recommend you include some comments in the file that warn about problems resulting from errors.

Again, the update.conf file is run on every host, including the server. The cfagent command is smart enough to copy the files locally (instead of over the network) when running on the configuration server. Several variables are defined in the control section and then used in the copy section. You can accomplish variable substitution with either the $(var) or ${var} sequence:

```
1 control:
2  actionsequence  = ( copy tidy )
3  domain          = ( mydomain.com )
4  workdir         = ( /var/cfengine )
5  policyhost      = ( server.mydomain.com )
6  master_cfinput  = ( /usr/local/var/cfengine/inputs )
7  cf_install_dir  = ( /usr/local/sbin )
8
9 copy:
10      $(cf_install_dir)/cfagent    dest=$(workdir)/bin/cfagent
11                                   mode=755
12                                   type=checksum
13
14      $(cf_install_dir)/cfservd    dest=$(workdir)/bin/cfservd
15                                   mode=755
16                                   type=checksum
17
```

```
18      $(cf_install_dir)/cfexecd    dest=$(workdir)/bin/cfexecd
19                                   mode=755
20                                   type=checksum
21
22      $(master_cfinput)            dest=$(workdir)/inputs
23                                   r=inf
24                                   mode=644
25                                   type=binary
26                                   exclude=*.lst
27                                   exclude=*~
28                                   exclude=#*
29                                   server=$(policyhost)
30                                   trustkey=true
31
32 tidy:
33   $(workdir)/outputs pattern=* age=7
```

Line 5: You should replace the string server.mydomain.com with the hostname of your configuration server.

Line 6: This is the directory on the master configuration server that contains the master configuration files. It requires the admit entry in cfservd.conf.

Line 7: This is the location, on every server, of the cfengine binaries.

Line 23: This specifies that you should copy the source directory recursively to the destination directory with no limit on the recursion depth.

Line 25: This option is misleading at first. It specifies that you should compare any local file byte by byte with the master copy to determine if an update is required.

Line 29: This option causes the files to be retrieved from the specified server.

Line 33: This command in the tidy section removes any outputs/ directory files that have not been accessed in the last seven days.

The permissions (modes) on each file are checked on each run even if the file already exists.

Framework for cfagent.conf

The cfagent.conf file is the meat of the cfengine configuration. You should make any change on any system using this file (or files imported from this file, as demonstrated in later chapters). We'll keep the sample cfagent.conf file simple for demonstration purposes; don't use it as is in a real-world scenario.

If you call any scripts from cfengine and those scripts produce any output, that output will be displayed (when executed interactively) or logged and e-mailed (when executed from `cfexecd`). Executing `cfagent` every hour is typical, so any scripts should produce output only if there is a problem or if something changed and the administrator needs to be notified:

```
1 control:
2 actionsequence  = ( files directories tidy disable processes )
3 domain          = ( mydomain.com )
4 timezone        = ( EDT EST )
5 access          = ( root )
6 # Where cfexecd sends reports
7 smtpserver      = ( mail.mydomain.com )
8 sysadm          = ( root@mydomain.com )
9
10 files:
11 /etc/passwd mode=644 owner=root action=fixall
12 /etc/shadow mode=600 owner=root action=fixall
13 /etc/group  mode=644 owner=root action=fixall
14
15 directories:
16 /tmp mode=1777 owner=root group=root
17
18 tidy:
19 /tmp recurse=inf age=7 rmdirs=sub
20
21 disable:
22 /root/.rhosts
23 /etc/hosts.equiv
24
25 processes:
26 "cfservd" restart "/var/cfengine/bin/cfservd"
27 "cfexecd" restart "/var/cfengine/bin/cfexecd"
```

Line 2: The `actionsequence` command is very important and easy to overlook. You must list each section that you wish to process in this variable. If you add a new section but forget to add it to this list, it will not be executed.

Line 4: Cfengine will make sure the system is configured with one of the time zones in this list.

Line 10: This section checks the ownership and permissions of a few important system files and fixes any problems it finds.

Line 15: This section checks the permissions on the /tmp/ directory and fixes them, if necessary. It also creates the directory if necessary.

Line 18: This section removes everything from the /tmp/ directory that has not been accessed in the past seven days. It removes only subdirectories of /tmp/—not the directory itself.

Line 21: These files are disabled for security reasons, and renamed if found. If they are executable, the executable bit is unset.

Line 25: This section verifies that the cfservd and cfexecd daemons are running and starts them if they are not.

Creating the Configuration Server

The configuration server contains the master copy of the cfengine configuration files. It also processes those configuration files on a regular basis as all the client systems do. The server must run a properly configured cfservd so the client systems can retrieve the master configuration from the system.

The configuration server needs a special place to keep the master cfengine configuration files. In this example, that directory is /usr/local/var/cfengine/inputs/. It could be any directory, but not /var/cfengine/inputs/ because the master host copies the files to that directory when executing, just like every other host.

Like all systems, the server should also run cfexecd either as a daemon or from cron (or, even better, both).

Now we'll discuss generating server keys. You need to run cfkey on the server system to create its public- and private-key files. These files will be in the /var/cfengine/ppkeys/ directory (or /var/lib/cfengine2/ppkeys on Debian and its derivatives) and will be named localhost.priv and localhost.pub.

The server also needs each client's public key in the appropriate file, based on the client's IP address as described in the next section. You can populate the file automatically upon the first connection of a remote host, similar to the way most SSH clients prompt the user to store a remote SSH server's host key for validation during subsequent connections. The TrustKeysFrom value in cfservd.conf and the TrustKey value in copy statements control server and client trust settings. We believe that trusted initial key exchange is a good idea, so we'll use that technique throughout this book.

CFENGINE AND ROOT PRIVILEGES

Cfengine does not require root privileges. The demonstration configuration files in this chapter perform operations that require root privileges, such as enforcing restrictive permissions on the `/etc/passwd` file.

You are encouraged to run cfengine as a nonprivileged user. The `cfagent` program defaults to the `~/.cfagent` directory instead of `/var/cfengine` for a working directory. The privileges needed are entirely dependent on the actions taken in the configuration files.

Preparing the Client Systems

Each client system is relatively simple to configure. Once you install the cfengine binaries, you need to generate the host's public and private keys (as discussed in this section). You also must copy the `update.conf` file from the master server manually and place it in `/var/cfengine/inputs/`. Once this file is in place, you should manually run `cfagent` to download the remaining configuration files and complete system configuration. We'll explore automated ways to handle this later on, but for now we're keeping things simple.

Each client should run `cfexecd` either as a daemon or from cron. You probably also want to run `cfservd` on each client to allow remote execution of `cfagent` using `cfrun`. Assuming automatic `cfexecd` execution has already configured in the `cfagent.conf` file on the server, these daemons will be started after the first manual execution of `cfagent`. From that point on, cfengine will be self-sustaining. It will update its own config files, and you can even use it to change its own behavior. For example, you can configure initialization scripts to start cfengine, change its schedule using `cfexecd` (from `cfagent.conf`), or even upgrade the cfengine binaries themselves.

You need to run `cfkey` on each client system before you run `cfagent` for the first time. This creates `localhost.priv` and `localhost.pub` in `/var/cfengine/ppkeys/`. You don't need to copy the central server's public key to the client manually. If the server's IP address is `10.1.1.1`, then when `cfagent` is run and it sees the `trustkey=true` entry in the `copy` section of `update.conf`, it will copy the server's public key to `root-10.1.1.1.pub` in the `/var/cfengine/ppkeys/` directory on the client. From that point on, the key is trusted and expected not to change for the host at IP `10.1.1.1`. If the key does change, by default cfengine will break off communications with the host when the key validation fails.

Debugging cfengine

When you are trying to get cfengine up and running, you will probably face a few problems. Network problems are common, for example, when you transfer configuration files from the master server and initiate `cfagent` execution on remote systems with `cfrun`.

For any network problems, you should run both the server (`cfservd`) and the client (`cfrun` or `cfagent`) in debugging mode. You can accomplish this by using the `-d2` command-line argument. For `cfservd`, this switch not only provides debugging output, but it also prevents the daemon from detaching from the terminal.

When you are trying something new in your `cfagent.conf` file, you should always try it with the `--dry-run` switch to see what it would do without making any actual changes. The `-v` switch is also useful if you want to see what steps `cfagent` is taking. If it is not doing something you think it should be doing, the verbose output will probably tell you why.

If you are making frequent changes or trying to get a new function to work properly, you probably want to be able to run `cfagent` repeatedly on demand. By default, `cfagent` will not do anything more frequently than once per minute. This helps prevent both intentional and accidental denial-of-service attacks on your systems.

You can eliminate this feature for testing purposes by placing this line in the `control` section of `cfagent.conf`:

```
IfElapsed = ( 0 )
```

You might also find it helpful to run only a certain set of actions by using the `--just` command-line option. For example, to check only on running processes, you can run the command `cfagent --just processes`.

Creating Sections in cfagent.conf

Cfengine 2.2.7 offers 34 possible sections in `cfagent.conf`; we'll cover some of these sections in this chapter, some later in the book, and some not at all. For additional information, refer to the comprehensive reference manual on the cfengine web site at `http://www.cfengine.org/`. Plus, read Brendan Strejcek's blog post about picking and choosing from the available cfengine feature set: "Cfengine Best Practices" (find it at `http://meta-admin.blogspot.com/2005/12/cfengine-best-practices.html`). It is also prominently displayed on the `http://www.cfwiki.org` site.

Every section can contain one or more class specifications. For example, the `files` section could be:

```
files:
   /etc/passwd

   any::
      /etc/group
   redhat::
      /etc/redhat-release
   solaris::
      /etc/vfstab
```

Both /etc/passwd and /etc/group will be processed on all hosts (because the default group is any when none is specified). In addition, the /etc/redhat-release file will be checked only on systems running Red Hat Linux, and the /etc/vfstab will be checked only if the operating system is Sun Microsystems' Solaris.

You can use the period (.) to "and" groups together, whereas you can use the pipe character (|) to "or" groups together. The exclamation character (!) can invert a class and parentheses (()) can group classes. Here is a complex example:

```
files:
   (redhat|solaris).!Mon::
      /etc/passwd
```

In this case, the /etc/passwd file will be checked only if the operating system is Red Hat Linux or Solaris and today is not a Monday.

Using Classes in cfagent.conf

You can use the classes section to create user-defined classes, as we described earlier. Determine a system's class membership using a shell command:

```
classes:
   X11R6 = ( '/usr/bin/test -d /usr/X11R6' )
```

If the command returns true (exit code 0), this machine will be a member of that class (for the current run). You can also define classes to contain specific hosts or any hosts that are members of another existing class:

```
classes:
   critical = ( host1 host2 web_servers )
```

Here are a few more possibilities that you could place in the classes section:

```
classes:
   notthis = ( !this )
   ip_in_range = ( IPRange(129.0.0.1-15) )
   ip_in_range = ( IPRange(129.0.0.1/24) )
```

The copy Section

The copy section is one of the most commonly utilized in cfengine. Cfengine can copy files between mounted filesystems on the client (whether local filesystems or remote shares), as well as from remote cfengine servers (systems running cfservd).

Cfengine copy operations prevent corruption or other errors in copied files by first copying the file to a file named *file*.cfnew on the local filesystem (where *file* is the name of the file being copied), then renaming the file quickly into place. This technique wards off problems resulting from partially copied or corrupted files being left in place (due to full disk, network interruption, system error, and so on).

You can choose from many configurable parameters for the copy section, but here are some used for the common scenario of copying an entire directory tree of programs:

```
copy:
      debian.i686::
            /usr/local/masterfiles/ganglia-3.0.7-debian-i686
                              dest=/usr/pkg/ganglia-3.0.7
                              mode=755
                              recurse=inf
                              owner=root
                              group=root
                              type=checksum
                              server=masterhost.mydomain.com
                              encrypt=true
```

This copy action is performed only on hosts with both the debian and i686 classes defined. Cfengine sets these classes automatically based on system attributes. Take advantage of these automatic classes to ensure that binary files of the correct type are copied.

The directory /usr/local/masterfiles/ganglia-3.0.7-debian-i686 is copied from a remote host named masterhost.mydomain.com. The remote host needs to run the cfservd daemon with access controls that allow the client system to access the source files. The setting recurse=inf specifies that the source directory be copied recursively—which means to recurse infinitely and copy all subdirectories, as well as all the files in the source directory. Instead of inf, you can specify a number to control the recursion depth.

The `copy` type is set to `checksum`, which means that an MD5 checksum is computed for each source and destination file, and that the destination file is updated only if the checksums don't match.

The owner and group are set to `root` on the destination, regardless of the ownership settings on the source files. We recommend explicitly setting ownership of copied files in every `copy` section, for security reasons. We set all the copied files to have the mode `755`, which will be executable for all users but writable only by the owner.

We set the network communications to be encrypted with the setting `encrypt=true`, because we like to keep all our administrative traffic private.

The directories Section

The `directories` section checks for the presence of one or more directories. Here is an example:

```
directories:
   /etc mode=0755 owner=root group=root syslog=true
      inform=true
   /tmp mode=1777 owner=root group=root define=tmp_created
```

If either directory does not exist, it will be created. The section will also check for permissions and ownership and correct them, if necessary. In this example, the administrator will be informed (through mail or the terminal) and a syslog entry will be created if the `/etc/` directory does not exist, or if it has incorrect permissions and/or ownership.

For the `/tmp/` directory, the class `tmp_created` is defined if the directory was created. You could then use this class in another section of the configuration file to perform certain additional tasks.

The disable Section

The `disable` section causes cfengine to disable any number of files. This simple section disables two files that you probably don't want around because they allow access to the `root` account through Remote Shell (RSH) using only the source IP address as authentication:

```
disable:
   /root/.rhosts
   /etc/hosts.equiv
```

If either of these files exists, the section will disable it by renaming it with a `.cfdisabled` extension. It will also change the permissions to `0600`, so you could use it to disable executables. At one point, for example, Solaris had a local root exploit with

the `eject` command. Until there was a patch available, you could have disabled that command using the following sequence:

```
disable:
   solaris::
      /usr/bin/eject inform=true syslog=true
```

This would not only disable the command, but it also would inform the administrator and make a log entry using syslog.

Suppose you want to remove `/etc/httpd/conf/httpd.conf` if it exists and create a symbolic link pointing to the master file `/usr/local/etc/httpd.conf`. The following command sequence can accomplish this task:

```
disable:
   /etc/httpd/conf/httpd.conf type=file define=link_httpd_conf
links:
   link_httpd_conf::
      /etc/httpd/conf/httpd.conf -> /usr/local/etc/httpd.conf
```

The `disable` section would remove the file only if it is a normal file (and not a link, for example). If the file is disabled, the `link_httpd_conf` class will be defined. Then, in the `links` section, a symbolic link will be created in its place.

Remember that cfengine does not execute these sections in any predefined order. The `actionsequence` setting in the `control` section controls the order of execution. So, for this example, make sure the file is disabled before the symbolic link is created:

```
control:
   actionsequence = ( disable links )
```

Managing the proper `actionsequence` order is something of an art form. As your cfengine configuration files grow in size, number, and complexity, you'll usually find both advantages and disadvantages to a particular ordering in your `actionsequence`. Once you find the order that works best in most cases, you'll want to keep the ordering in mind as you write configurations that set classes to trigger other actions, and try to line up your dependencies accordingly. To see what we mean, skip ahead to the section on NTP client configuration in Chapter 7. When NTP configuration files are copied:

1. A class called `restartntpd` is defined.

2. In the `shellcommands` section, the NTP daemon is restarted with a startup script.

3. If the `shellcommands` section doesn't come after the `copy` section in the `actionsequence`, then this sequence of events can't happen as planned.

You can also use the `disable` section to rotate log files. The following sequence rotates the web-server access log if it is larger than 5MB and keeps up to four files:

```
disable:
    /var/log/httpd/access_log size=>5mbytes rotate=4
```

The editfiles Section

The `editfiles` section can be the most complex section in a configuration file. You can choose from approximately 100 possible commands in this section. These commands allow you to check and modify text files (and, in a few cases, binary files).

We won't go into great detail on using `editfiles` because you should use this section rarely. We generally prefer copying complete files to clients. This way, we can keep our systems' file contents properly in a revision-control system. You can use a script to create the proper file contents for different systems in a central location, after which classes based on system attributes (e.g., Debian vs. Solaris, or web server vs. mail server) can copy the correct file into place.

That said, you'll encounter some situations where doing a direct file modification on clients can be appropriate and useful, such as maintaining a message of the day or updating files that were previously updated by a manual process and that differ from system to system. The examples here are for demonstration purposes.

Here is an example `editfiles` section:

```
editfiles:
    { /etc/crontab
        AppendIfNoSuchLine "0 * * * * root /usr/local/sbin/cfexecd -F"
    }
```

This command adds the specified line of text to `/etc/crontab`. This makes sure that cron runs `cfexecd` every hour. You also might want to make sure that other hosts can access and use the printers on your printer servers:

```
editfiles:
    PrintServer::
        { /etc/hosts.lpd
            AppendIfNoSuchLine "host1"
            AppendIfNoSuchLine "host2"
        }
```

In your environment, perhaps a standard port is used for another purpose. For example, you might want to rename port 23 to `myservice`. To do this, you could change its label in `/etc/services` on every host:

```
editfiles:
  { /etc/services
    ReplaceAll "^.*23/tcp.*$" With "myservice 23/tcp"
  }
```

If you are using `inetd` and want to disable the `telnet` application, for example, you could comment out those lines in `/etc/inetd.conf`:

```
editfiles:
  { /etc/inetd.conf
    HashCommentLinesContaining "telnet"
    DefineClasses "modified_inetd_conf"
  }
processes:
  modified_inetd_conf::
    "inetd" signal=hup
```

Any line containing the string `telnet` will be commented out with the # character (if not already commented). If you make such a change, the `inetd` process is sent the HUP signal in the `processes` section.

The files Section

The `files` section can process files (and directories) and check for valid ownership and permissions. It can also watch for changing files. Here is a simple example:

```
files:
  /etc/passwd mode=644 owner=root group=root action=fixall checksum=md5
  /etc/shadow mode=600 owner=root group=root action=fixall
  /etc/group mode=644 owner=root group=root action=fixall

  web_servers::
    /var/www/html r=inf mode=a+r action=fixall
```

We accomplish several tasks with these entries. On every system, the `files` section checks and fixes the ownership and permissions on `/etc/passwd`, `/etc/shadow`, and `/etc/group`. It also calculates and records the MD5 checksum of `/etc/passwd`.

On any system in the class `web_servers`, the permissions on `/var/www/html/` are checked because that is the standard web-content directory across all hosts at our site. Your site might (and probably will) differ. The directory is scanned recursively and all files and directories are made publicly readable. The execute bits on directories will also be set

according to the read bits; because we requested files to be publicly readable, directories will also be publicly executable.

The `checksum` option requires a little more explanation. The file's checksum is stored, so the administrator will be warned if it changes later. In fact, the administrator will be warned every hour unless you configure cfengine to update the checksum database. In that case, the administrator will be notified only once, and then the database will be modified. You can enable this preference by adding the following command in the `control` section:

```
control:
   ChecksumUpdates = ( on )
```

Here are some other options you might want to use in the `files` section:

`links`: You can set this option to `traverse` to follow symbolic links pointing to directories. Alternatively, you can set it to `tidy` to remove any dead symbolic links (links that do not point to valid files or directories).

`ignore`: You can specify the `ignore` option multiple times. Cfengine version 2 requires a pattern or a simple string, but not a regular expression. Cfengine version 3 repairs this inconsistency (look for more information on cfengine 3 at the end of this chapter). Any file or directory matching this pattern is ignored. For instance, `ignore="^\."` would ignore all hidden files and directories.

`include`: If any `include` options are listed, any files must match one of these regular expressions in order to be processed.

`exclude`: Any file matching any of the `exclude` regular expressions will not be processed by cfengine.

`define`: If the directives in this `files` section result in changes being made to any listed files, a new class will be defined. You can also list several classes, separated by colons.

`elsedefine`: Similar to define, but here new classes are defined if *no changes* are made to the listed files. You can also list several classes, separated by colons.

`syslog`: When set to `on`, cfengine will log any changes to the system log.

`inform`: When set to `on`, cfengine will log any changes to the screen (or send them by e-mail if `cfagent` is executed by `cfexecd`).

The links Section

With the `links` section, `cfagent` can create symbolic links:

```
links:
    /usr/tmp -> ../var/tmp
    /var/adm/messages  ->! /var/log/messages
    /usr/local/bin +> /usr/local/lib/perl/bin
```

In this example, the first command creates a symbolic link (if it doesn't already exist) from `/usr/tmp` to `../var/tmp` (relative to `/usr/tmp`).

The second command creates a symbolic link from `/var/adm/messages` to `/var/log/messages` *even if there is already a file* located at `/var/adm/messages`. The bang (`!`) overrides the default safety mechanism built into cfengine around creating symbolic links.

The third command creates one link in `/usr/local/bin/` pointing to each file in `/usr/local/lib/perl/bin/`. Using this technique, you could install applications in separate directories and create links to those binaries in the `/usr/local/bin/` directory.

The `links` section offers plenty of possible options, but they are rarely used in practice so we won't cover them in this book. See the cfengine reference manual for more information.

The processes Section

You can monitor and manipulate system processes in the `processes` section. Here is an example from earlier in this chapter:

```
processes:
    "cfservd" restart "/var/cfengine/bin/cfservd"
    "cfexecd" restart "/var/cfengine/bin/cfexecd"
```

For the `processes` section, cfengine runs the `ps` command with either the `-aux` or `-ef` switch (as appropriate for the specific system). This output is cached and the first part of each command in the `processes` section is interpreted as a regular expression against this output. If there are no matches, the `restart` command is executed.

You can specify the following options when using the `restart` facility: `owner`, `group`, `chroot`, `chdir`, and/or `umask`. These affect the execution environment of the new process as started by the `restart` command.

You can also send a signal to a process:

```
processes:
    "httpd" signal=hup
```

This signal would be sent on every execution to any processes matching the regular expression httpd, so you probably don't want to use it as is. It is also possible to specify limits on the number of processes that can match the regular expression. If you wanted to ensure there were no more than ten httpd processes running at any given time, for example, you would use this code:

```
processes:
    "httpd" action=warn matches=<10
```

The shellcommands Section

For some custom and/or complex operations, you will need to execute one or more external scripts from cfengine. You can accomplish this with the shellcommands section. Here is an example:

```
shellcommands:
    all::
        "/usr/bin/rdate -s ntp1" timeout=30
    redhat.Hr02_Q1::
        "/usr/local/sbin/log_packages" background=true
```

On all systems, the rdate command is executed to synchronize the system's clock. Cfengine terminates this command in 30 seconds if it has not completed. On systems running Red Hat Linux, a script runs between 2:00 a.m. and 2:15 a.m. to log the currently installed packages. This command is placed in the background and cfengine does not wait for it to complete. This way, cfengine can perform other tasks or exit while the command is still running.

You can specify the following options to control the environment in which the command is executed: owner, group, chroot, chdir, and/or umask.

If these scripts want to access the list of currently defined classes, they can look in the CFALLCLASSES environment variable. Each active class will be listed, separated by colons.

Scripts, by default, are ignored when cfengine is performing a dry run (with --dry-run specified). You can override this setting by specifying preview=true. The script should not, however, make any changes when the class opt_dry_run is defined.

Using cfrun

The cfrun command allows you to execute cfagent on any number of systems on the network. It requires a configuration file in the current directory named cfrun.hosts (or a file specified with the -f option). The cfrun.hosts file can be as simple as this:

```
domain=mydomain.com
server.mydomain.com
client1.mydomain.com
client2.mydomain.com
```

Apart from the domain setting, this file is just a list of every host, including the configuration server. You can also have the output logged to a series of files (instead of being displayed to the screen) by adding these options to the top of the file:

```
outputdir=/tmp/cfrun_output
maxchild=10
```

This code tells `cfrun` to fork up to ten processes and place the output for each host in a separate file in the specified directory. You can normally run `cfrun` without arguments. If you do want to specify arguments, use this format:

```
cfrun CFRUN_OPTIONS HOSTS -- CFAGENT_OPTIONS -- CLASSES
```

`CFRUN_OPTIONS` is, quite literally, optional, and can contain any number of options for the `cfrun` command. Next, you can specify an optional list of hostnames. If some hostnames are specified, only those hosts will be contacted. If no hosts are specified, every host in the `cfrun.hosts` file will be contacted.

After the first --, you must place all options you want to pass to the `cfagent` command run on each remote system. After the second -- is an optional list of classes. If some classes are specified, only hosts that match one of these classes will execute `cfagent` (although each host is contacted because each host must decide if it matches one of the classes).

Looking Forward to Cfengine 3

Cfengine 3 has been in the design phase for several years. It is a complete rewrite of the cfengine suite, but more important, it involves a new way of thinking about system management.

Cfengine 3 is built around a concept called "Promise Theory." This concept might sound difficult to grasp, but it's actually quite intuitive. With cfengine 3, you'll describe the *desired state* of your systems instead of the *changes* to your systems. The desired state is expressed as a collection of promises, and in the words of the cfengine author Mark Burgess, allows us to focus on the good instead of the bad.

The Cfengine.org web site has a thorough introduction to cfengine 3, as well as source code to the current snapshot of cfengine 3 development: `http://www.cfengine.org/cfengine3.php`.

We encourage you to familiarize yourself with the next evolutionary steps in cfengine for two reasons:

1. Familiarity with the new concepts and syntax will make it easier to migrate from version 2 to version 3 when the time comes.

2. Experimenting with the current feature set and implementation allows you to suggest enhancements or bug fixes. Making suggestions helps the people working on cfengine 3 and "gives back" to the people who gave us cfengine in the first place.

Using cfengine in the Real World

In this chapter, we covered the core concepts of cfengine and demonstrated basic usage with a collection of artificial configuration files. This information arms you with the knowledge you need to work through the remainder of this book.

The use of demonstration configuration files and imaginary scenarios ends here. Throughout the rest of this book, we will operate on a real-world infrastructure that we build from scratch. Every configuration setting and modification that we undertake will be one more building block in the construction of a completely automated and fully functional UNIX infrastructure.

C H A P T E R 5

■ ■ ■

Bootstrapping
a New Infrastructure

How would you feel if you were offered the opportunity to install and configure all the systems and applications at a new startup company? On one hand you'd probably be pleased, because you would finally be able to fix all the things you've criticized about other infrastructures. On the other hand you'd be apprehensive, because you have only one chance to build it before it goes into production, and you might be blamed (with good reason) if things don't work well.

We would expect most people to admit to feelings on both ends of the spectrum. If you're anything like us, you'll be thrilled to have the opportunity! Radically changing the design of your cfengine master, system upgrade, and patching procedures is easy *before* the systems and procedures are put into use. Once you've been deploying and updating systems using automated means, reorganizing and rebuilding the automation framework is much more difficult and risky. You can rebuild some or all of your environment later if you employ development and/or staging systems, but any time you spend now on planning and design will certainly pay off in the form of fewer headaches later.

We'll show you how to build systems for a fictional startup company called "campin. net," a purveyor of quality camping equipment. The project: to deploy a web-based application that utilizes data over NFS. The web servers run Linux, and the NFS server hosts run Solaris.

One of the major goals of this project is to rapidly deploy and configure the systems hosting the site, as well as to deploy application and OS updates soon after the need becomes apparent. Deploying and managing systems by automated means will meet these goals.

You should set up a management infrastructure before you image or deploy any systems, because you need to be prepared to manage the new hosts from the very beginning. We're using cfengine for system management, so we need to have cfengine configured and running on each host upon completion of the imaging process.

We must perform this sequence of steps to configure a fully functional cfengine infrastructure:

1. Manually install a Linux system to be used as the central cfengine host.

2. Create a "master" directory structure on the central cfengine host. This central directory structure is where cfengine configuration files as well as UNIX/Linux configuration files and binaries will be copied from. We'll illustrate an example layout that we'll continue to develop in later chapters.

3. Populate the directory structure with cfengine configuration files. You'll use these configuration files to perform initial client configuration, keep clients up to date with the configuration files hosted on the central cfengine host, and start up the cfengine daemons.

4. Choose a trust model for cfengine key distribution. We're using a model that resembles the way key trusts are done with SSH hosts—where we trust a host's cfengine keys the first time we see them. After the initial exchange, we'll use that trusted key (and *only* that key) to verify that host's identity.

This procedure is highly technical and fast paced. We recommend reading through it once before undertaking it at your site.

Installing the Central cfengine Host

We decided to use virtualization for the initial master system, specifically VMware Server. This decision makes sense because once the guest system is working, we can bring it into the datacenter and run it on anything from a laptop running Windows or Linux to a dedicated VMware ESX platform. VMware probably *isn't* what we'd choose for enterprise-wide virtualization, mainly because of the license fees. This isn't a book on virtualization, however, so we won't go into detail on virtualization technologies.

We installed a 32-bit Debian GNU/Linux 4.0 ("Etch") system as a VMware guest. We're not going to cover manual Debian installations, although we should mention that on the "software selection" screen, we selected "standard system" and none of the other software collections. This kept the base system very small. From there we manually installed the `cfengine2` package as the `root` user:

```
# apt-get install cfengine2
```

The default Debian `cfengine2` package installation does not put any files in place in the `/var/lib/cfengine2/inputs` directory. We won't place any there yet either. Cfengine supplies a pre-configuration script feature that we'll use to bootstrap systems once our master repository is set up.

Setting Up the cfengine Master Repository

The design goals of our cfengine master repository are:

- *Simplicity*: The framework should be simple and intuitive. It should follow UNIX conventions as frequently as possible.

- *Transparency*: The design should provide for easy inspection and debugging.

- *Flexibility*: The system will be used for purposes we can't imagine now, so we need to make it extensible. UNIX is built on the idea that users know better than designers what their needs are, and we'll honor that tradition.

It pays to think about how you want to lay out your cfengine master host's files before you get started. Once you've put it into use and have many different types of systems copying files from many different locations on the server, reorganizing things will be much trickier.

We're using a Debian host as our master, so we'll use the `/var/lib/cfengine2` directory as our base for our cfengine "masterfiles" directory. There is nothing special about the name "masterfiles" except that it's used in the cfengine sample configurations. We use the convention as well.

First, as the `root` user, we create the directory `/var/lib/cfengine2/masterfiles`, then the directories we need inside it:

```
# mkdir /var/lib/cfengine2/masterfiles
# cd /var/lib/cfengine2/masterfiles
# mkdir PROD DEV STAGE
# mkdir -p PROD/inputs PROD/repl/root/etc/passwd PROD/repl/root/etc/group
# mkdir -p PROD/repl/root/etc/shadow PROD/repl/root/etc/sudoers
# mkdir -p PROD/repl/root/etc/ntp PROD/repl/root/etc/fai PROD/repl/root/etc/
# mkdir -p PROD/repl/admin-scripts
```

In the preceding commands, we've created directories that mimic files inside the UNIX filesystem, underneath `/var/lib/cfengine2/masterfiles/PROD/repl/root`. The `repl` directory is meant for files that are pulled or "replicated" from the master system out to client systems. Using a tree that resembles where files live on client systems is very intuitive to SAs.

In later chapters we will illustrate how to use the `STAGE` and `DEV` directory trees for testing new cfengine settings in a testing and/or staging environment. For now we're only populating the `PROD` branch since we have only production hosts (and only one at that!).

Creating the cfengine Config Files

As previously mentioned, cfengine looks for configuration files in a directory called `inputs`, usually located at `/var/cfengine/inputs` (or `/var/lib/cfengine2/inputs` on Debian). In our initial configuration, as well as our configurations going forward, `cfagent.conf` will be made up entirely of `import` statements. You'll find that `import` statements in cfengine resemble includes in most programming languages—the imported file is read in and used as if it were content in the original file. For the cfengine configuration file sections such as `control` and `classes`, `cfagent.conf` will import files in the `control` and `classes directories`. We'll create the directories first (as `root`):

```
# cd /var/lib/cfengine2/masterfiles/PROD/inputs
# mkdir -p hostgroups tasks/os tasks/app tasks/misc control classes modules filters
```

Here's how these directories will function:

- `cfagent.conf` will import other files for all of its configuration entries. These files will be in the `control`, `classes`, `filters` and `groups` directories.

- Modules will be placed in the `modules` directory. In cfengine, you use modules to extend functionality (we illustrate modules in the Appendix).

- At the end of `cfagent.conf` are what we call "hostgroup" imports. Each `hostgroup` file comprises further imports of files in the `tasks` directory. The files in the `tasks` directory contain actual work for cfengine, such as copying files, killing processes, and editing files. Each task accomplishes a single goal, such as distributing `ntp.conf` and restarting the `ntpd` daemon when the change is made.

We'll separate tasks that focus on applications from OS-specific tasks or tasks that focus on the core OS. We do this by splitting "task" files between the `apps`, `os`, and `misc` subdirectories. This division will make it easier to remove entire applications from the cfengine configuration once they are unnecessary. We're trying to build a system that will manage the systems at our site for the next five to ten years (or more).

The cf.preconf Script

Cfengine provides initial bootstrap capability through the `cf.preconf` script, which gets the system to the point where it can parse and utilize the cfengine configuration files.

In our case we use it to ensure the host has generated a cfengine key, to generate initial configuration files, and to create the required directories. Cfengine looks for the `cf.preconf` script before it parses the configuration files, and executes it. Cfengine feeds

it one argument: the hard class of the system (e.g., `linux`). Our site's `cf.preconf` script doesn't make use of this argument.

The `cf.preconf` script is entirely optional, and you can write it in any language supported on the local system. No problems or errors will result from a site choosing not to implement it.

Here's our version for our new environment, explained section by section. We assume basic shell-scripting expertise on the part of the reader and focus on the intent of the script instead of the mechanics of each line. And comments intended for future maintainers of the script are always useful:

```sh
#!/bin/sh
###################################################
#
# Be very, very afraid when changing this script ...it's meant to get
# cfengine to a state where it'll work properly. It's intended for
# such things as getting the basic files and directories in place for
# cfengine.
# This script should be fast, simple and efficient.
# Errors/mistakes are to be avoided at all costs.
#
###################################################


PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/pkg/cfengine/sbin:/opt/csw/sbin:/opt/csw/bin
export PATH

CFWORKDIR=/var/cfengine

if [ -f /etc/debian_version ]; then
    CFWORKDIR=/var/lib/cfengine2
fi

create_update_conf() {

# create basic update.conf, meant to update all the config
# files from the master
```

This next line starting with the `cat` command will place the script contents up until the line starting with "EOF" into the two cfengine config files `update.conf` and `cfagent.conf` (thanks to the `tee` invocation). You'll find it convenient to embed a file in a script because it means you can maintain one file instead of two:

```
cat <<EOF  | tee $CFWORKDIR/inputs/update.conf $CFWORKDIR/inputs/cfagent.conf
##############################################################
# Created by cf.preconf, and you shouldn't ever see this comment
# in \$CFWORKDIR/inputs/update.conf or
# \$CFWORKDIR/inputs/cfagent.conf
#
# The fact that you're reading it now it means that this host wasn't
# able to pull files from the cfengine server (as of this writing:
# goldmaster.campin.net).
#
# Please investigate. Normally it means you're having cfengine key
# or access control issues.
##############################################################
control:

    solaris|solarisx86::
        cf_base_path = (  /opt/csw/sbin )
        workdir = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )

    debian::
        cf_base_path = ( /usr/sbin )
        workdir = ( /var/lib/cfengine2 )
        client_cfinput = ( /var/lib/cfengine2/inputs )

    redhat::
        cf_base_path = ( /usr/pkg/cfengine/sbin )
        workdir = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )

    !debian.!redhat.!(solaris|solarisx86)::
        # take a best guess on the path for other hosts, since we use
        # any:: below to keep cfengine procs alive on all hosts
        # We could use debian.32_bit.sunos_sun4u:: instead but we'd rather
        # at least try on other hosts
        cf_base_path   = ( /usr/pkg/cfengine/sbin )
        workdir         = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )
```

Up to this point in the update.conf and cfagent.conf files, we simply define some variables, i.e., the locations of important files across the different platforms we support. How these are used will become clear further in the file.

The `actionsequence` controls which cfengine actions are executed and the order in which they're run:

```
any::
    actionsequence = ( copy.IfElapsed0

                                        links.IfElapsed0
                                      processes.IfElapsed0

    )
```

Because `cf.preconf` generated these `update.conf` and `cfagent.conf` files, we're worried only about the basics: `copy`, `links`, and `processes`. Each action's appended `IfElapsed0` works around cfengine's built-in denial-of-service protections (or "spamming," as it's called in the cfengine documentation). Cfengine uses a time-based locking mechanism that prevents actions from being executed unless a certain minimum time has elapsed since the last time they were executed. The `IfElapsed0` modifier sets the lock time to zero, disabling it. We put this there simply to be sure that the copies are attempted each time `cfagent` is run. We want to do everything possible to get the system past the initial bootstrap phase.

The `SplayTime` variable becomes important as our site grows:

```
    domain              = ( campin.net )
    policyhost          = ( goldmaster.campin.net )
    master_cfinput  = ( /var/lib/cfengine2/masterfiles/PROD/inputs )
    SplayTime           = ( 0 )
```

If hundreds of systems start `cfagent` at the same time, contention or load issues might result when all systems simultaneously request a file from the cfengine master system. If this number is nonzero, `cfagent` goes to sleep after parsing its configuration file and reading the clock. Every machine will go to sleep for a different length of time, which is no longer than the time you specify in minutes. For our bootstrap configuration, we don't need the `SplayTime` functionality, and we set it to zero.

Items declared under the global `ignore` section affect the `copy` and `links` actions of our initial bootstrap configuration files:

```
ignore:
    # RCS and subversion directories will be ignored in all copies
    RCS
    .svn
```

The `ignore` section prunes directories, meaning that any Subversion, RCS, or CVS (which uses RCS for its base functionality) metafiles will not be copied. We don't have such things in place yet, but intend to implement them later.

We always want the cfengine daemons to be running. Here's where our earlier per-platform variables come into play—on each of our three future UNIX/Linux platforms, this single `processes` stanza will properly start the cfengine daemons:

```
processes:
        any::
                "cfexecd" restart "\$(cf_base_path)/cfexecd"
                "cfservd" restart "\$(cf_base_path)/cfservd"
                "cfenvd" restart "\$(cf_base_path)/cfenvd"
```

Note that we need to escape the dollar sign on the $(cf_base_path) variables in order to keep the cf.preconf shell script from attempting to expand them as shell variables. This process continues for the rest of the embedded cfengine configuration inside cf.preconf.

We make sure here that the cfengine base directory always has a bin directory with a working cfagent symlink inside it:

```
links:
    # for some reason Solaris x86 doesn't always consider itself
    # "solaris", so we add "solarisx86". Newer cfengine versions
    # seem to have fixed this, but we're paranoid. :)
    redhat|solaris|solarisx86::
        /var/cfengine/bin/cfagent -> \$(cf_base_path)/cfagent
```

The Debian cfengine2 package already sets up a symlink from /var/lib/cfengine2/bin to /usr/bin so no changes are required, and the debian class is left out of this section.

On all cfengine clients, everything in master_cfinput (which is /var/lib/cfengine2/ masterfiles/PROD/inputs) and everything in its subdirectories is copied to $(workdir)/ inputs/ on every host:

```
copy:

        \$(master_cfinput)/          dest=\$(workdir)/inputs/
                                     r=inf
                                     mode=700
                                     type=binary
                                     exclude=*~
                                     exclude=#*
                                     exclude=*,v
                                     purge=true
                                     server=\$(policyhost)
                                     trustkey=true
                                     encrypt=true
```

This is the main focus of our initial bootstrap cfengine configuration: simply to get the latest configuration files from the master. Once that is done, the cfengine daemons are started.

Remember that the `actionsequence` is:

1. `copy`

2. `links`

3. `processes`

This means that even though `copy` comes after `links` and `processes` in the configuration file, the `actionsequence` runs `processes` last. This is the correct and intended order.

This ends the embedded cfengine configuration file(s):

```
EOF
```

Now we make sure important directories are in place, and that they're protected from snooping by non-`root` users:

```
}

DIRS="$CFWORKDIR/ppkeys $CFWORKDIR/inputs $CFWORKDIR/bin"
for dir in $DIRS
do
    [ -d $dir ] || mkdir -p $dir && chmod 700 $dir
done
```

This next code is site-specific:

```
CURRENT_CF_RH_VER=cfengine-2.2.7

if [ -f /etc/redhat-release ]; then
    # need a critical symlink
    if [ -d /usr/pkg/$CURRENT_CF_RH_VER -a ! -d /usr/pkg/cfengine ]; then
        ln -s /usr/pkg/$CURRENT_CF_RH_VER /usr/pkg/cfengine
    fi
fi
```

We ensure that on Red Hat Linux we have a symlink to the latest installed version of cfengine (see Chapter 6 for a discussion of Kickstart, the automated installation system for Red Hat Linux). Our installation scripts will take care of the same thing, but extra checking is prudent.

If the `update.conf` file is missing, generate it using the `create_update_conf()` subroutine from earlier in the script:

```
if [ ! -f $CFWORKDIR/inputs/update.conf ]; then
    create_update_conf
fi
```

If the host's cfengine key is missing, generate it. This is the final line of `cf.preconf`:

```
[ -f $CFWORKDIR/ppkeys/localhost.priv ] || cfkey
```

The update.conf file

When the `cfexecd` scheduler daemon executes `cfagent`, the `update.conf` configuration file is read before `cfagent.conf`. It contains private classes and variables from the rest of the cfengine configuration files, and ensures that the local configuration files are up to date. We also utilize it to make sure certain important actions happen early, such as enforcing the existence and permissions of certain directories, confirming that cfengine daemons are running, and performing some basic cleanup.

Here's our campin.net `update.conf` file, starting with the warning comments for future maintainers:

```
#######
#
# This script distributes the configuration, a simple file so that,
# if there are syntax errors in the main config, we can still
# distribute a correct configuration to the machines afterwards, even
# though the main config won't parse. It is read and run just before
# the main configuration is parsed.
#
#######
#
# You almost NEVER want to modify this file, you risk breaking
# everything if it's not correct
#
#######
```

The following section is the same content we embedded in the `cf.preconf` file:

```
control:

    solaris|solarisx86::
        cf_base_path = (   /opt/csw/sbin )
        workdir  = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )

    debian::
        cf_base_path = ( /usr/sbin )
        workdir = ( /var/lib/cfengine2 )
        client_cfinput = ( /var/lib/cfengine2/inputs )

    redhat::
        cf_base_path = ( /usr/pkg/cfengine/sbin )
        workdir = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )

    !debian.!redhat.!(solaris|solarisx86)::
        # take a best guess on the path for other hosts
        cf_base_path = ( /usr/pkg/cfengine/sbin )
        workdir = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )
```

We have additional actions in our `update.conf` file, compared to the one embedded in `cf.preconf`:

```
    any::

        actionsequence  = (
            links
            directories
            copy
            processes.IfElapsed0
            tidy
        )
```

We'll describe the new sections as we get to them. Always double check that your ordering is still correct any time that you add or remove items from the `actionsequence`.

We use this `addinstallable` line to let cfengine know that we might define a new class at runtime:

```
Addinstallable = ( hupcfexecdandcfservd )
```

Cfengine is a one-pass interpreter, and it allocates internal space for classes that it determines it might need during the current run.

We set `SplayTime` to 5, meaning that `cfagent` will sleep a random interval between zero and five minutes before taking any action:

```
domain      = ( campin.net )
policyhost = ( goldmaster.campin.net )
fileserver  = ( goldmaster.campin.net )

master_cfinput  = ( /var/lib/cfengine2/masterfiles/PROD/inputs )
SplayTime          = ( 5 )
```

This `copy` is the same cfengine configuration-file directory copy that we embedded in `cf.preconf`. It keeps our local cfengine configuration files up to date, by copying the master `inputs` tree every time `cfagent` runs:

```
copy:
      $(master_cfinput)/
          dest=$(workdir)/inputs/
          r=inf
          mode=700
          type=checksum
          ignore=RCS
          ignore=.svn
          owner=root
          group=root
          ignore=*,v
          purge=true
          server=$(policyhost)
          trustkey=true
        encrypt=true
```

We use the `directories` action to enforce the existence and proper permissions of important cfengine directories:

```
directories:

    any::
        $(workdir)/backups mode=700 owner=root group=root
        $(workdir)/inputs mode=700 owner=root group=root
        $(workdir)/outputs mode=700 owner=root group=root
        $(workdir)/ppkeys mode=700 owner=root group=root
        $(workdir)/cfengine/sums mode=700 owner=root group=root
        $(workdir)/tmp mode=700 owner=root group=root

    !debian::
        $(workdir)/bin mode=700 owner=root group=root
```

If you later decided to allow access to a UNIX group made up of cfengine administrators, you could easily change the group permissions and ownership to match this selective access. For security reasons, don't allow global access to these directories (even read access).

The `cfexecd` daemon stores output from `cfagent` in the $(workdir)/outputs directory and e-mails the output if configured to do so. We don't want the directory to grow without bounds, so we delete files older than seven days:

```
tidy:
    any::
        $(workdir)/outputs pattern=* age=7
```

We have the same process monitoring for cfengine daemons as before:

```
processes:

    any::
        "cfexecd" restart "$(cf_base_path)/cfexecd"
        "cfservd" restart "$(cf_base_path)/cfservd"
        "cfenvd" restart "$(cf_base_path)/cfenvd"

    hupcfexecdandcfservd::
        "cfexecd" signal=hup inform=true
        "cfservd" signal=hup inform=true
```

```
links:
    !debian::
        $(workdir)/bin/cfagent -> $(cf_base_path)/cfagent
            define=hupcfexecdandcfservd
```

But now when the hupcfexecdandcfservd class is defined in the links section, cfengine will send a HUP signal to the cfexecd and cfservd processes. It works like this:

1. If the symlink in $(workdir)/bin/cfagent pointing to $(cf_base_path)/cfagent is missing, cfagent will create it.

2. When the symlink is created, the class hupcfexecdandcfservd is defined.

3. The processes section has a stanza for systems with the hupcfexecdandcfservd class defined. Now cfagent knows it's supposed to run this section, so it sends a HUP signal to the cfexecd and cfservd processes.

You'll notice that this looks quite a bit like the update.conf and cfagent.conf that cf.preconf creates. The main difference is that the ones created by cf.preconf were intended only to get updated cfengine configuration files pulled to the client. Any time spent making other changes to the system from cf.preconf is unnecessary because the latest policies for the site will be pulled from the master and applied to the local system. We'll continuously update the main configuration files on the cfengine master, but cf.preconf will see infrequent edits. You'll need less maintenance this way.

The cfagent.conf file

As you'll remember from Chapter 4, cfagent.conf is the main cfengine configuration file. The cfagent program looks there for its important settings, and expects the entirety of your policy declarations to be made inside it (or inside files imported from cfagent.conf using the import directive, to be covered shortly).

Here is our cfagent.conf:

```
####################################################
#
# The cfengine config file organization is simple.
#
# Only the main cfengine config files go into $(workdir)/inputs.
# These are:
#
```

```
#   update.conf
#   cfservd.conf
#   cfagent.conf
#   cf.preconf
#   cfrun.hosts
#
# Everything else is organized in subdirectories beneath this. We have
# these subdirectories so far:
#
#   tasks/
#   tasks/os
#   tasks/app
#   hostgroups/
#   modules/
#   classes/
#   control/
#   filters/
#
# In tasks/os I have files like cf.motd, used to configure our
# site-wide message of the day file contents. That is clearly a task.
#
# In hostgroups/ I have files which are meant specifically for
# importing tasks which that class of host needs to perform. If you
# have an action that only needs to be performed on a single host, you
# should still define a hostgroup (and think of it as a role), but
# only have the single host inside it. In the event that you have to
# perform the tasks against more hosts, you simply add the new hosts
# to the group.
#
# No actual cfengine actions (copy, processes, etc) should be taken
# inside the hostgroups/* files, they should use the files in tasks/
# to actually make changes on hosts.
#
# With this configuration we can easily see the tasks that make up the
# configuration of all our hostgroups, and also easily inspect what
# each task does in a short amount of time (because the files are
# generally small and limited in scope).
#
```

```
# So in a nutshell: all hosts should be in one or more hostgroups,
# which will cause that host to import the relevant hostgroup file(s),
# which will then import all the tasks it needs to configure it the
# way we want that type of host configured.
#
################

import:
    # define some classes we'll be needing
    classes/cf.main_classes

    # setup the main cfengine variables, the "control" section
    control/cf.control_cfagent_conf

    # the control section that manages cfexecd
    control/cf.control_cfexecd

    #
    # Stuff for every single host. Be careful with what you put in
    # this one.
    #
    any::        hostgroups/cf.any
```

First, notice that the file has more lines of comments than lines of cfengine configuration. This is because `cfagent.conf` is where most SAs will start with cfengine modifications, and we want future users of our system to be constantly reminded of how things are organized at our site. Document your configuration as if you're setting it up for others, and assume that you won't be personally available for questions. Doing so will help you bring on new staff quickly, as well as let you take time off without constantly answering basic questions.

Second, notice that our `cfagent.conf` is made up entirely of `import` statements. You could put your entire configuration into a single `cfagent.conf` file, but that's generally considered a bad idea. If we had done that at our last employers, `cfagent.conf` would have been more than 30,000 lines long. Making sense of such a huge configuration file is too difficult, and you'd face extra overhead by having all systems process the configuration directives meant for themselves as well as all other hosts. If we split it up from the start, you won't have to go through the pain of reorganization later. In the world of system administration, many (if not most) temporary configurations end up being permanent. It pays to do things correctly and in a scalable manner from the start.

Here are the contents of `classes/cf.main_classes`:

```
classes:

    weekend    = ( Saturday Sunday )
    weekdays  = ( Monday Tuesday Wednesday Thursday Friday )
    weekday    = ( weekdays )

    web_servers    = (    rmlamp
            etchlamp
    )

    have_svc = ( "/usr/bin/test -d /service" )
```

(You can refer to our explanation of cfengine classes in Chapter 4.) Right now we
have very few classes, but that will change. We'll eventually have multiple files with many
different types of classes.

Here are the contents of `control/cf.control_cfagent_conf`:

```
control:

    solaris|solarisx86::
        cf_base_path    = (   /opt/csw/sbin )
         workdir            = ( /var/cfengine )
        client_cfinput  = ( /var/cfengine/inputs )

    debian::
        cf_base_path   = ( /usr/sbin )
        workdir            = ( /var/lib/cfengine2 )
        client_cfinput = ( /var/lib/cfengine2/inputs )

    redhat::
        cf_base_path    = ( /usr/pkg/cfengine/sbin )
         workdir            = ( /var/cfengine )
        client_cfinput  = ( /var/cfengine/inputs )

    !debian.!redhat.!(solaris|solarisx86)::
        cf_base_path   = ( /usr/pkg/cfengine/sbin )
        workdir            = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )
```

```
any::
    addinstallable     = (    hupcfexecdandcfservd    )
    AllowRedefinitionOf = ( branch )
```

We will use the $(branch) variable for easy switching between production and non-production variants of our cfengine configurations in later chapters. For now, we're simply setting up the mechanism to enable this:

```
domain                     = ( campin.net )
policyhost              = ( goldmaster.campin.net )
fileserver                 = ( goldmaster.campin.net )

 maxage                   = ( 7 )

TrustKeysFrom   = ( goldmaster.campin.net )

branch                     = ( PROD )
 master_cfinput    = ( /var/lib/cfengine2/masterfiles/$(branch)/inputs )
master                     = ( /var/lib/cfengine2/masterfiles/$(branch) )
master_etc     = ( /var/lib/cfengine2/masterfiles/$(branch)/repl/root/etc )

SplayTime     = ( 5 )
IfElapsed     = ( 4 )
```

We'll set all hosts to use the PROD directory path, then selectively set it again on particular hosts to use either the DEV or STAGE directory. Cfengine will consider it an error to redefine a variable if the variable isn't listed in AllowRedefinitionOf.

The IfElapsed parameter controls the minimum time that must elapse before an action in the actionsequence will execute again. Remember that you can also set it on a per-action basis as we did in cf.preconf with the syntax actionSequence = (copy.IfElapsed15).

The repository setting defines a directory where you can place backup and junk files:

```
repository       = ( $(workdir)/backups )
```

These are files affected by editfiles, disable, copy, and links. When cfengine replaces or disables a file, the previous contents of that file are placed in the repository directory.

The Editfilesize setting is a safety check that the editfiles action uses:

```
Syslog                   = ( on )
SyslogFacility = ( LOG_DAEMON )
Editfilesize       = ( 10024576 )
```

By default, cfengine won't attempt to edit a file larger than 10,000 bytes. We set that value to a much larger size, because some of our config files will exceed the default size by a considerable margin. We don't make much use of editfiles, but it comes in handy occasionally.

These are security-related settings:

```
SecureInput                = ( on )
NonAlphaNumFiles  = ( on )
FileExtensions             = ( o a ) # etc
SuspiciousNames        = ( .mo lrk3 lkr3 )
```

If SecureInput is set to on, cfagent will import only files owned by the user running the program. When NonAlphaNumFiles is set to on, cfengine will automatically disable any nonalphanumeric files it finds during file sweeps, under the assumption that these files might be deliberately concealed. The FileExtensions directive will generate warnings when it finds any directories with the extensions listed. The assumption here, again, is that an attempt is being made to hide directories. The SuspiciousNames directive generates warnings for any files found with the listed file extensions during file sweeps. Note that the file sweeps happen only when cfengine scans a directory in connection with a command such as files, tidy, or copy.

The actionsequence here has comments around it in the file, once again intended for future cfengine configuration-file maintainers:

```
#
# Keep editfiles before processes, since you might want to
# do something like edit a daemon's config file with
# editfiles then HUP it with processes.
#
# disable is before copy since sometimes we delete a
# symlink before copying in a real file
#
actionsequence = (
    directories
    disable
    copy
    editfiles
    links
    files
    processes
    shellcommands
)
```

Not all the actions are used immediately, but in Chapter 7 we plan to demonstrate all of the listed actions as we begin setting up our example infrastructure.

We define some variables that are used in later file copies, namely `fileserver` and `policyhost`. We keep them separate so we can split the file copies between two different cfengine master hosts without having to modify every copy task. We would need to change the variables only in `control/cf.control_cfagent_conf`.

Here are the contents of `control/cf.control_cfexecd`:

```
control:
    any::

        # When should cfexecd in daemon mode wake up the agent?
        schedule    = ( Min00_05 Min20_25 Min40_45 )
```

(You can find more about the `schedule` setting in Chapter 4.) We think that running `cfagent` from `cfexecd` every 20 minutes will be the proper frequency for our new site. Many sites run only once an hour, but we like the increased frequency because we will use cfengine to make sure that important processes are running. The actions or tasks that don't need to be performed this often will be set to run only when certain time- or day-based classes are set, such as `Hr01.Min00_05` to make an action happen between 1:00 a.m. and 1:05 a.m. The reason for the range is that we've set a five-minute `SplayTime` and `cfagent` will run at a random time inside that range. We will have extensive examples of time-based classes in later chapters.

Here are the configuration directives pertaining to the e-mail sent from `cfexecd`; they are self-explanatory:

```
        EmailMaxLines   = ( 5000 )     # max lines of output to email
        EmailFrom         = ( cfengine@campin.net )
        sysadm             = ( ops@example.org )    # where to mail output
        smtpserver        = ( smtp.campin.net )    # mail relay over SMTP
```

This final line in `control/cf.control_cfexecd` tells `cfexecd` where to find the `cfagent` binary:

```
        cfrunCommand  = ( "$(cf_base_path)/cfagent" )
```

■**Warning** The `cfrunCommand` variable is meant for the `cfservd` daemon according to the cfengine documentation, but in the past we've found that `cfexecd` doesn't function correctly without this variable set up. This might be fixed in recent versions of cfengine, but it's another area where we're paranoid, so we keep fixes in place indefinitely.

Here are the contents of `hostgroups/cf.any`:

```
# imported by ALL hosts, careful with this one
#
# disted by cfengine, don't edit locally
import:
        any::
                tasks/os/cf.motd
                tasks/os/cf.cfengine_cron_entries
```

This file is also made up only of imports—for a very important reason. Once we have spent some time developing and maintaining our cfengine configurations, we'll end up with many actions being done to particular groups of hosts. Using this layout, we'll have a file for each role or group of hosts performing similar duties that shows the collection of tasks that are performed for that role or group. This is an easy way to document what we do to configure a certain type of host. The alternative is a very long list of imports for each role or group directly out of `cfagent.conf`, and eventually the file gets so large it is difficult for humans to read and understand easily.

We already have two task files used on all hosts at our site. We want every host that we set up at our site to have some special cron entries, as well as a standard message of the day to greet users when they log in.

The cf.motd Task

We think it's important to maintain a standard message-of-the-day file for all hosts that we administer. We like to greet authorized users with a message about where to find help and greet unauthorized users with a warning.

We also like each host to have entries specific to itself. In cfengine we set up support for local message-of-the-day contents using a file at the location `/etc/motd-local`.

We'll now list the contents of `tasks/os/cf.motd`. Here we test for the existence of the local `motd` file. If it exists, it alters the behavior of this task:

```
classes:  # synonym groups:
    have_etc_motd_local    = ( FileExists(/etc/motd-local) )
```

In the `control` section we define some variables:

```
control:
    masterfile       = ( $(master_etc)/motd-master )
    local_masterfile = ( /etc/motd-master )
    local_message    = ( /etc/motd-local )
```

In the copy section we copy the site-wide motd file from the fileserver host:

```
copy:
    any::
        $(masterfile)
            dest=$(local_masterfile)
            mode=755
            owner=root
            group=root
            type=checksum
            server=$(fileserver)
            encrypt=true
```

In the editfiles section, cfengine will evaluate the edit actions in the three groups if the file /etc/motd-local exists on the system:

```
editfiles:
    have_etc_motd_local::

        { /etc/motd

            BeginGroupIfFileIsNewer "$(local_message)"
                EmptyEntireFilePlease
                InsertFile "$(local_masterfile)"
                InsertFile "$(local_message)"
                PrependIfNoSuchLine "This system is running $(class):$(arch)"
            EndGroup

            BeginGroupIfFileIsNewer "$(local_masterfile)"
                EmptyEntireFilePlease
                InsertFile "$(local_masterfile)"
                InsertFile "$(local_message)"
                PrependIfNoSuchLine "This system is running $(class):$(arch)"
            EndGroup

            BeginGroupIfNoLineContaining "campin.net"
                EmptyEntireFilePlease
                InsertFile "$(local_masterfile)"
                InsertFile "$(local_message)"
                PrependIfNoSuchLine "This system is running $(class):$(arch)"
            EndGroup
        }
```

The first two groups will trigger when either of the `motd-local` or `motd-master` files are newer than the `/etc/motd` file. If that's the case, you must update `/etc/motd`.

You accomplish the update by merging the two files together, with the `motd-master` file inserted first. This behavior resembles the `make` utility's functionality.

The third group is triggered when the string "campin.net" isn't found in `/etc/motd`. We know that our `motd-master` file contains that string, so this is a way to update `/etc/motd` for the first time. After that, `/etc/motd` will be newer than `motd-master` and `motd-local`, and won't be re-edited until one of the two input files is updated.

This section is similar to the previous `edit` section, but will be evaluated if the file `/etc/motd-local` does not exist on the system:

```
!have_etc_motd_local::

    { /etc/motd

        BeginGroupIfFileIsNewer "$(local_masterfile)"
            EmptyEntireFilePlease
            InsertFile "$(local_masterfile)"
            PrependIfNoSuchLine "This system is running $(class):$(arch)"
        EndGroup

        BeginGroupIfNoLineContaining "campin.net"
            EmptyEntireFilePlease
            InsertFile "$(local_masterfile)"
            PrependIfNoSuchLine "This system is running $(class):$(arch)"
        EndGroup
    }
```

The master `motd` file is inserted when it's newer than `/etc/motd`, or if the string "campin.net" isn't found in `/etc/motd`.

The file `/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/motd-master` is copied from the `fileserver` host to `/etc/motd-master` on all systems.

These are the contents of `motd-master`:

```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        You are on a private campin.net system.
        Unauthorized use is prohibited, all activity is monitored.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
## SA Staff: for local-only additions here, put content in /etc/motd-local ##
```

This is a good way to send a standard message on login, but still allow for local modifications. A great secondary benefit is that once a host is imaged and configured to use

cfengine, it should automatically update this file. This means that we should be greeted with our site's message of the day upon our first login to a newly imaged host, giving us instant validation of the host's cfengine configuration.

The cf.cfengine_cron_entries Task

The second entry in `hostgroups/cf.any` is `tasks/os/cf.cfengine_cron_entries`, and it has these contents:

```
control:
```

```
        addinstallable          = (      hupcron )
```

In the `control` section we specify a definable class that we define later in the file, if and when the edits defined are performed. The following edits are performed only if the required entries aren't there:

```
editfiles:

    debian::
      { /var/spool/cron/crontabs/root
       AutoCreate
       AppendIfNoSuchLine "31 23 * * * /usr/sbin/cfagent"
       DefineClasses "hupcron"
      }

    solaris|solarisx86::
      { /var/spool/cron/crontabs/root
       AppendIfNoSuchLine  "31 23 * * * /opt/csw/sbin/cfagent"
       DefineClasses "hupcron"
      }

    redhat::
      { /var/spool/cron/root
       AutoCreate
       AppendIfNoSuchLine "31 23 * * * /usr/pkg/cfengine/sbin/cfagent"
       DefineClasses "hupcron"
      }
```

With these cron entries, we make sure all our hosts run the `cfagent` command once per day. This isn't meant to handle the "normal" scheduled runs; for that we will use `cfexecd`. These cron entries are meant to restore our systems to a working state when

all other `cfagent` startup and scheduling mechanisms fail. Our `update.conf` file specifies the startup of the cfengine daemons, so if we manage somehow to kill off our cfengine daemons or fail to start them upon boot, the daemons will start back up again within 24 hours when `cfagent` is called again from cron.

When cron files are updated on Linux, the cron program will notice the file update and reread the files automatically:

```
shellcommands:

        (solaris|solarisx86).hupcron::
                # when the crontab is updated on solaris, restart cron
                # (we only install solaris 10)
                " /usr/sbin/svcadm restart cron" timeout=30
```

Based on this behavior, we never actually do anything with the `hupcron` class on Linux. The cron daemon on Solaris has no such feature, and needs to be restarted. That's what we do in the `shellcommands` section when `hupcron` is defined.

The `cf.cfengine_cron_entries` task ensures that `cfagent` is run at least once per day. This will get `cfagent` to update the inputs directories and start up `cfexecd`, `cfenvd`, and `cfservd` if they're not already running.

These cron entries are our emergency measure intended to get cfengine back up and running if all other measures fail and we either don't know that it's broken or we can't access the host in order to fix it manually.

cfservd.conf

One other important file needs to be created: `cfservd.conf`. We need to allow client systems to pull files from the *goldmaster* host, and also set up the access and proper path to the `cfagent` program for remote invocations through `cfrun`. This file is used on all systems, because we choose to run `cfservd` on all systems.

Here are the contents of `cfservd.conf`:

```
control:
    any::
        domain       = ( campin.net )
        AllowUsers  = ( root )
```

Trust in cfengine is done by private keys, so the `access` directive gives `cfservd` access only to the key for the named users. For this to happen, the `root` key must already be in place, so there's a race condition when a new host is imaged (although for a short period).

■**Warning** You should be aware that the only way to prevent a malicious user from spoofing the key of a new host (and the only trusted user, `root`) is to turn off key trust and to install keys using some other mechanism outside the cfengine trusted key exchange. See the online cfengine documentation for further details. We utilize cfengine initial trusts in this book, because the added security from manual or out-of-band cfengine key distribution is negligible.

The `ifElapsed` feature is also built into `cfservd`:

```
ifElapsed    = ( 1 )
```

You use it to prevent clients from forcing `cfservd` to reread its configuration too often. The default is one hour.

Once again we define the location of important directories and binaries:

```
ExpireAfter                = ( 15 )
MaxConnections       = ( 5 )
MultipleConnections = ( true )
LogAllConnections     = ( true )
TrustKeysFrom          = ( 192.168.1 )

solaris|solarisx86::
    cf_base_path       = ( "/opt/csw/sbin" )
    cfrunCommand   = ( "/opt/csw/sbin/cfagent" )

debian::
    cf_base_path     = ( "/usr/sbin" )
    cfrunCommand = ( "/usr/sbin/cfagent" )

redhat::
    cf_base_path       = ( "/usr/pkg/cfengine/sbin" )

!debian.!redhat.!(solaris|solarisx86)::
    # take a best guess on the path for other hosts, since I use
    # any:: below to keep cfengine procs alive on all hosts
    cf_base_path        = ( "/usr/pkg/cfengine/sbin" )
    cfrunCommand   = ( "/usr/pkg/cfengine/sbin/cfagent" )
```

Eventually we plan to run many hosts at our site, and often a single host will make more than one connection to our master host. We want this limit to be higher than we think we'll need:

```
goldmaster::
    MaxConnections    = ( 500 )
```

Prior experience shows that this number needs to be approximately twice the number of actual cfengine clients that will simultaneously connect to the cfservd daemon.

Note that only our *goldmaster* host is sharing files via cfservd. That's where our fileserver and policyhost variables currently point. All the other hosts simply allow cfagent invocations via the cfrun tool:

```
admit:
    # execute the true location of cfagent remotely using cfrun,
    # symlinks don't count
    any::
            /var/cfengine/bin/cfagent                      *.campin.net 192.168.1
            /usr/pkg/cfengine-2.2.7/sbin/cfagent    *.campin.net 192.168.1
            /usr/pkg/cfengine/sbin/cfagent             *.campin.net 192.168.1
            /usr/sbin/cfagent                              *.campin.net 192.168.1
            /opt/csw/sbin/cfagent                          *.campin.net 192.168.1

    goldmaster::
        # Grant access to all hosts in *.campin.net
        /var/lib/cfengine2/masterfiles          192.168.1 *.campin.net
            root=192.168.1 encrypt=true
```

Ready for Action

This is a good time to step back and look at what we've accomplished, because we've covered quite a bit in this chapter.

We've set up and configured a fully functional cfengine master, ready for use by client systems to pull configuration files. We have also set up a small amount of system configuration, namely editing of cron files and message-of-the-day files. We now have what we need to manage newly installed hosts at our site using cfengine.

C H A P T E R   6

■ ■ ■

Setting Up Automated Installation

It is critically important that you use unattended operating-system installation tools for every system you deploy. If you were to load the OS manually, you would rarely (if ever) get the same configuration on all your systems. It would be nearly impossible to configure your systems properly from that point forward.

Automated installation systems offer several key benefits:

- When all systems are initially configured in an identical manner, automated system-configuration tools such as cfengine don't have to account for varying initial system state. This minimizes the complexity of the automation system itself.

- Many systems can be installed and deployed at once, even by a junior SA or data-center technician. Once the automated installation system is configured, very little work is required to add new installation clients.

- An automated procedure for OS installation can be considered a form of documentation. A manual OS installation process might be documented, but you have no proof that the final system state is really the result of the documented steps.

- You can use a backup of the automation system to deploy many systems in a new location with confidence that the resulting systems are properly installed.

We hope this chapter removes any fear you might have around setting up automated installation systems. We're well aware that vendor documentation for systems such as Sun's Custom JumpStart can be intimidating. We firmly believe that even sites with a very small number of UNIX or Linux systems need to use automated OS installation techniques for every new host.

We aren't going to explain how such automated installation systems *could* work; we're going to show you how they *really* work. We're going to set up automated installation systems to deploy real systems, and document the procedure from start to finish.

Introducing the Example Environment

We're going to deploy systems into a completely new environment for the fictional startup company we described in Chapter 5. We'll be running x86-based Red Hat Enterprise Linux 5.2 systems hosting a web application, with UltraSPARC-based Solaris 10 systems sharing application data over NFS. In addition, we'll deploy x86-based Debian GNU/Linux 4.0 systems to provide infrastructure services (e.g., cfengine, DNS, Network Time Protocol, etc.).

We made the decision to run two different Linux distributions:

1. We find Debian GNU/Linux to be easy to administer, making it a good fit for infrastructure roles.

2. The (fictional) business decided to use Red Hat for support reasons, so we'll use Red Hat on systems where vendor support is important (e.g., our web servers).

The network in our new environment is flat: a single subnet utilizing a private (RFC1918) IP range. We'll initially use this single subnet for imaging as well as production service. We won't discuss network-administration details such as routing and switching.

To deploy our three different OS platforms, we'll create three different system-imaging servers:

- To image our Debian systems, we'll use FAI, or Fully Automatic Installation (see `http://www.informatik.uni-koeln.de/fai/`).

- We'll use Sun's Custom JumpStart to image our Solaris machines (see `http://docs.sun.com/app/docs/doc/817-5506/jumpstartoverview-4?a=view`).

- We'll use Kickstart to image our Red Hat systems (see `http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.2/html/Installation_Guide/ch-kickstart2.html`).

Each of our imaging systems will utilize postinstallation scripts that we develop. These scripts will cause the system to utilize our new campin.net cfengine infrastructure from Chapter 5. All our new systems will be booted from the network, and during the imaging process they will have cfengine installed and configured to use our cfengine master system. Cfengine will handle *all* system configuration from the very first bootup of our hosts.

FAI for Debian

FAI is an unattended imaging system built for Debian Linux. Here's a definition from the FAI Guide at `http://www.informatik.uni-koeln.de/fai/fai-guide/ch-intro.html`:

> *FAI is a noninteractive system to install a Debian GNU/Linux operating system on a single computer or a whole cluster. You can take one or more virgin PCs, turn on the power and after a few minutes Linux is installed, configured, and running on the whole cluster, without any interaction necessary. Thus, it's a scalable method for installing and updating a cluster unattended with little effort involved. FAI uses the Debian GNU/Linux distribution and a collection of shell and Perl scripts for the installation process. Changes to the configuration files of the operating system can be made by cfengine, shell, Perl, and Expect scripts.*

Note the mention of cfengine scripts used during the installation process. Those familiar with cfengine can easily understand FAI configuration and usage. FAI also has the concept of classes at its core, and uses assignment to classes and the definitions assigned to those classes to determine how a host is installed and configured.

Here are the steps required to set up FAI from scratch and image our first Debian system:

1. Install a Debian system manually for use as the installation server.

2. Install the FAI packages along with dependent packages on the new system.

3. Configure FAI.

4. Run `fai-setup` to create the NFS root filesystem for installation clients.

5. Configure network booting for installation clients.

6. Customize the installation process for all systems, as well as special configuration particular to our first installation client.

7. Boot and install our first FAI client system.

Once again we find ourselves in need of a host, in order to configure other hosts. Our cfengine master system (named *goldmaster*) will function as our FAI installation host. This host is running the current stable branch, Debian 4.0. The IP address of this host on our example network is `192.168.1.249`.

If you encounter any problems with the examples and commands in this section, refer to the online FAI documentation here: `http://www.informatik.uni-koeln.de/fai/fai-guide/ch-inst.html#s-faisetup`. By the time you read this, Debian 5.0 ("Lenny") will surely be out, and there's a chance that you'll need to update this procedure.

Installing and Configuring the FAI Packages

Install the needed packages by using `apt-get` or `aptitude` to install the `fai-quickstart` metapackage:

```
# aptitude install fai-quickstart
```

This code will install all the needed packages, such as `dhcp3-server` and `tftpd-hpa`, as well as the `fai-client` and `fai-server` packages.

Now that you have the required packages, edit `/etc/fai/make-fai-nfsroot.conf`. This file controls the creation of the nfsroot filesystem in `/srv/fai/nfsroot`. You need to make only these minor changes from the default:

```
NFSROOT_ETC_HOSTS="192.168.1.249 goldmaster"
SSH_IDENTITY=/home/nate/.ssh/id_dsa.pub
```

Be sure to substitute the proper values for your network. Here is the file in its entirety:

```
# these variables are only used by make-fai-nfsroot(8)
# here you can use also variables defined in fai.conf

# directory on the install server where the nfsroot for FAI is
# created, approx size: 250MB, also defined in bootptab or dhcp.conf
NFSROOT=/srv/fai/nfsroot

# Add a line for mirrorhost and installserver when DNS is not available
# on the clients. This line(s) will be added to $nfsroot/etc/hosts.
NFSROOT_ETC_HOSTS="192.168.1.249 goldmaster"

FAI_DEBOOTSTRAP="etch http://ftp.debian.org/debian"

# the encrypted (with md5 or crypt) root password on all install clients during
# installation process; used when log in via ssh; default pw is: fai
FAI_ROOTPW='FOOBARBAZ.E$djxB128U7dMkrFOOBARBAZ'

# this kernel package will be used when booting the install clients
KERNELPACKAGE=/usr/lib/fai/kernel/linux-image-2.6.18-fai-kernels_1_i386.deb

# location of a identity.pub file; this user can log to the install
# clients in as root without a password; only useful with FAI_FLAGS="sshd"
SSH_IDENTITY=/home/nate/.ssh/id_dsa.pub
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# following lines should be read only for most of you

FAI_DEBOOTSTRAP_OPTS="--exclude=dhcp-client,info"
```

The configuration for the FAI package (but not the configuration for installation clients) is stored in /etc/fai/fai.conf. We didn't change anything in fai.conf. Here is the complete file from our *goldmaster* system:

```
# /etc/fai/fai.conf -- configuration for FAI (Fully Automatic Installation)

# Access to Debian mirror via NFS mounted directory
# If FAI_DEBMIRROR is defined, install clients mount it to $MNTPOINT
#FAI_DEBMIRROR=yournfs debianmirror:/path/to/debianmirror

# LOGUSER: an account on the install server which saves all log-files
# and which can change the kernel that is booted via network.
# Configure .rhosts for this account and PAM, so that root can log in
# from all install clients without password. This account should have
# write permissions for /srv/tftp/fai. For example, you can use write
# permissions for the group linuxadm. chgrp linuxadm /srv/tftp/fai;chmod
# g+w /srv/tftp/fai. If the variable is undefined, this feature is disabled.
# Define it, to enable it, eg. LOGUSER=fai
LOGUSER=

# set protocol type for saving logs. Values: ssh, rsh, ftp
FAI_LOGPROTO=rsh

# the configuration space on the install server
FAI_CONFIGDIR=/srv/fai/config

# how to access the fai config space
# default if undefined here: nfs://`hostname`/$FAI_CONFIGDIR
# supported URL-types: nfs, file, cvs, cvs+ssh, svn+file, svn+http,...
#FAI_CONFIG_SRC=nfs://yourservername$FAI_CONFIGDIR

# the following variables are read only for most users

# mount point where the mirror will be mounted
MNTPOINT=/media/mirror

# the local configuration directory on the install client
FAI=/var/lib/fai/config
```

FAI uses apt-get to create the nfsroot filesystem. Once /etc/fai/fai.conf and /etc/fai/make-fai-nfsroot.conf are configured to your liking, run fai-setup:

```
# fai-setup
```

A lot of information will scroll by, but you need to look for these two lines that indicate success:

```
make-fai-nfsroot finished properly.
FAI setup finished.
```

If you don't see them, you'll need to troubleshoot your configuration. Most problems result from improper settings in /etc/fai/make-fai-nfsroot.conf or simply insufficient disk space on your host.

The nfsroot creation aspect of fai-setup is done when fai-setup invokes the make-fai-nfsroot command. In order to troubleshoot, you can call make-fai-nfsroot yourself with the -v flag to see more verbose (and useful) output. This is the best way to find out what caused the failure. Refer to the online FAI guide for up-to-date troubleshooting information.

Configuring Network Booting

We intend to boot our hosts from the network using PXE. PXE, which stands for Pre-eXecution Environment, is a method to boot computers using a network interface independent of any available storage devices or installed operating systems. Most network cards manufactured in the last several years support PXE boot.

DHCP is a standard method for assigning IP network information to hosts. DHCP servers can hand out the required information for PXE clients to boot from the network. The fai-quickstart metapackage installed software that we can use to boot network clients using PXE. These are the required Debian packages: tftp-hpa, tftpd-hpa, syslinux, dhcp3-common, and dhcp3-server. If any of these are missing from your system, install them using aptitude or apt-get.

Copy the sample dhcpd.conf file from the FAI examples directory into place for your DHCP server:

```
# cp /usr/share/doc/fai-doc/examples/etc/dhcpd.conf /etc/dhcp3/dhcpd.conf
```

The file on our *goldmaster* system looks like this after editing:

```
deny unknown-clients;
option dhcp-max-message-size 2048;
use-host-decl-names on;
```

```
subnet 192.168.1.0 netmask 255.255.255.0 {
    option routers 192.168.1.1;
    option domain-name "campin.net";
    option domain-name-servers 192.168.1.1;
    option time-servers 192.168.1.249;
    option ntp-servers 192.168.1.249;
    option tftp-server-name "goldmaster.campin.net";
    server-name goldmaster;
    next-server 192.168.1.249;
    filename "fai/pxelinux.0";
}
```

```
host etchlamp {hardware ethernet 00:0c:29:25:ea:c7;fixed-address etchlamp;}
```

One of the most important settings is the first line in the file: `deny unknown-clients`. This setting ensures you boot only hosts that are specifically configured to do so. We've gathered the Ethernet MAC address of our first Debian system to be installed with FAI, and put it in this configuration file at the end.

In addition, we placed this new host, named *etchlamp*, into our site's DNS. We already conveniently host DNS with a DNS-hosting provider, so we managed to avoid setting it up initially at our new site. We'll set up our own internal DNS in Chapter 7.

The `tftpd` daemon runs from the `inetd` super server, so make sure you have a line like this in `/etc/inetd.conf`:

```
tftp dgram udp wait root /usr/sbin/in.tftpd /usr/sbin/in.tftpd -s /srv/tftp
```

If you add it, be sure to HUP the running `inetd` process.

To configure an FAI install client, use the command `fai-chboot`. When you're using PXE, this invocation tells the install client to boot the install kernel and perform an installation during the next boot:

```
# fai-chboot -IFv etchlamp
```

Because the hostname is already in DNS (you can use your `/etc/hosts` file if you're completely lacking DNS at the start), and the host's Ethernet MAC address is in the `dhcpd.conf` file, `fai-chboot` can set up the proper PXE boot configuration file in `/srv/tftp/fai/pxelinux.cfg/`.

Customizing the Install Client

Now we can boot a host, but we'll want some customization before we attempt an installation. We'll want to be sure a web server is preconfigured on the first Debian host we image. We'll end up using it as an infrastructure web server for Subversion, Nagios, and other applications. We'll get to those applications in later chapters, but for now we'll just worry about getting Apache 2 up and running.

To define a new class of our own in FAI, create a script called `60-more-host-classes` and place it in the `/srv/fai/config/class` directory. This new script sets a class called `WEB` for our new host that denotes `webserver`:

```
#! /bin/bash

case $HOSTNAME in
    etchlamp)
        echo "WEB" ;;
esac
```

Setting a new class in FAI is as easy as creating the preceding script. That class is then used in other scripts within FAI that install packages, run scripts, and configure the system's disk drives. FAI's use of classes resembles the way cfengine uses classes.

■**Note**   The numbers prepended to the script names in the FAI script directory are used for the same purpose as the numbers in the names of run-control scripts such as those in `/etc/rc3.d/` on Red Hat, Debian, and Solaris systems. They're used to order the execution of scripts in a directory. Under FAI, though, the start of a file name contains no S or K—only a number.

We already have a `50-host-classes` file that is installed by default with FAI in the same directory, and it resembles the new `60-more-host-classes` file. You want to make sure your customizations are contained in discrete files as often as possible. When you later choose to build an FAI server automatically, you won't have to edit files programmatically. This means you'll only have to copy a new file into place, which is always less error-prone, and also means that the FAI authors' updates to the scripts don't need to be merged back into your copy of the file.

Also in the `/srv/fai/config/class/` directory is a file called `FAIBASE.var`. This file contains settings for all hosts installed using FAI, because FAI applies the `FAIBASE` class to all installation clients. Some variables in this file need modification: the time zone is wrong for our site, as is the keymap. You'll also need to change the `root` password from the default (in this same file) by putting a new MD5 or crypt entry in this file for the `ROOTPW` variable.

Here's the FAIBASE.var file after our modifications:

```
# default values for installation. You can override them in your *.var files

# allow installation of packakges from unsigned repositories
FAI_ALLOW_UNSIGNED=1

CONSOLEFONT=
KEYMAP=us-latin1

# Set UTC=yes if your system clock is set to UTC (GMT), and UTC=no
# if not.
UTC=yes
TIMEZONE=US/Pacific

# root password for the new installed linux system; md5 and crypt
# are possible
# pw is "fai"
ROOTPW='Ragbarfoo3f3Y'

# moduleslist contains modules that will be loaded by the new system,
# not during installation these modules will be written
# to /etc/modules
# If you need a module during installation, add it to $kernelmodules
# in 20-hwdetect.source. But discover should do most of this job
MODULESLIST="usbkbd ehci-hcd ohci-hcd uhci-hcd usbhid psmouse"
```

We've already decided that our new host *etchlamp* will belong to the WEB class. Let's set up a custom package list for the WEB class in a new file in the /srv/fai/config/ package_config/ directory. As you've probably guessed, FAI uses this directory to define the packages installed for classes of hosts. All hosts will by default use the FAIBASE package configuration, but our new host needs some additional packages.

Here are the contents of /srv/fai/config/package_config/WEB:

```
PACKAGES aptitude
apache2-utils
apache2.2-common
apache2-mpm-prefork
libapr1
libexpat1
libpq4
```

```
libsqlite3-0
libaprutil1
mime-support
libapache2-mod-php5
libxml2
php5
php5-common
```

This takes care of our wishes for the packages installed for the WEB class.

It is so easy to configure exactly which packages should go onto a system that we decided we wanted to modify the base system. Namely, we changed /srv/fai/config/package_config/FAIBASE to use postfix and syslog-ng instead of exim and sysklogd. We added these lines:

```
postfix openssl ssl-cert
syslog-ng
```

and we removed this line:

```
exim4
```

The next step is configuration of our first host's disk layout. We set up custom partitioning for the WEB class in the file /srv/fai/config/disk_config/WEB:

```
# <type> <mountpoint> <size in mb> [mount options]    [;extra options]
disk_config disk1
primary  /              150-      rw,errors=remount-ro ; -c -j ext3
logical  swap          400-500    rw
```

Finally, you want to make sure cfengine is configured properly and that it's pulling configuration files from the master system after installation. The first step is to make sure that cfengine daemons are started at boot time. We handle this by creating a cfengine script and placing it at /srv/fai/config/scripts/FAIBASE/50-cfengine:

```
#! /usr/sbin/cfagent -f

control:
   any::
   actionsequence = ( editfiles )
   EditFileSize = ( 30000 )
```

```
editfiles:
    any::
        { ${target}/etc/aliases
          AutoCreate
          AppendIfNoSuchLine "root: ops@example.org"
        }

        { ${target}/etc/default/cfengine2
          ReplaceAll "=0$" With "=1"
        }
```

The edit of the mounted root filesystem's `/etc/default/cfengine2` in this cfengine script changes the lines:

```
RUN_CFSERVD=0
RUN_CFEXECD=0
RUN_CFENVD=0
```

to these:

```
RUN_CFSERVD=1
RUN_CFEXECD=1
RUN_CFENVD=1
```

At boot time or if manually executed, the Debian cfengine2 init script will start the cfengine daemons only if the values of the `RUN_*` variables are set to `1`. This `50-cfengine` script also ensures that `/etc/aliases` contains a `root` alias before installation is complete.

Now we need to get the files `update.conf` and `cfagent.conf` in place for when cfengine starts up upon our new host's first boot. We'll use FAI's `fcopy` command to move the `update.conf` and `cfagent.conf` files into place during installation. We'll create `/srv/fai/config/scripts/FAIBASE/60-create-cf-config`, with these contents:

```
#!/bin/bash
error=0 ; trap "error=$((error|1))" ERR

fcopy etc/cfengine/update.conf
fcopy etc/cfengine/cfagent.conf

exit $error
```

The `fcopy` command works on files placed under `files/` in the FAI `config` directory, in a directory named after the file you need to copy. The files in the directory, which are named after FAI classes, contain the appropriate contents for hosts matching the class

contained in the file name. According to the FAI docs, if multiple classes match, then the class with the highest matching priority gets its file copied.

We're using the FAIBASE class because we want all hosts to get the basic `update.conf` and `cfagent.conf` files.

The contents of the identical `/srv/fai/config/files/etc/cfengine/cfagent.conf/` FAIBASE and `/srv/fai/config/files/etc/cfengine/update.conf/FAIBASE` files are:

```
control:

        solaris|solarisx86::
                cf_base_path    = (   /opt/csw/sbin )
                workdir             = ( /var/cfengine )
                client_cfinput  = ( /var/cfengine/inputs )

        debian::
                cf_base_path    = ( /usr/sbin )
                workdir             = ( /var/lib/cfengine2 )
                client_cfinput  = ( /var/lib/cfengine2/inputs )

        redhat::
                cf_base_path    = ( /usr/pkg/cfengine/sbin )
                workdir             = ( /var/cfengine )
                client_cfinput  = ( /var/cfengine/inputs )

        !(debian|redhat|solaris|solarisx86)::
                cf_base_path    = ( /usr/pkg/cfengine/sbin )
                workdir             = ( /var/cfengine )
                client_cfinput  = ( /var/cfengine/inputs )

        any::
                actionsequence  = (      copy.IfElapsed0
                                         links.IfElapsed0
                                         processes.IfElapsed0
                                         )

                domain          = ( campin.net )
                policyhost      = ( goldmaster.campin.net )
                master_cfinput  = ( /var/lib/cfengine2/masterfiles/PROD/inputs )
                SplayTime       = ( 0 )
```

```
ignore:
      # RCS/svn administrative stuff will be ignored in all copies
      RCS
      .svn

processes:

        any::
                "cfexecd" restart "$(cf_base_path)/cfexecd"
                "cfservd" restart "$(cf_base_path)/cfservd"
                "cfenvd" restart "$(cf_base_path)/cfenvd"

links:
        redhat|solaris|solarisx86::
                /var/cfengine/bin/cfagent      -> $(cf_base_path)/cfagent

        # debian already sets up a symlink from /var/lib/cfengine2/bin
        # to /usr/bin so no changes required on that platform

copy:

#
# Everything in /var/cfengine/masterfiles/inputs on the master
# _and_ everything in its subdirectories is copied to every host.
#
        $(master_cfinput)/            dest=$(workdir)/inputs/
                                       r=inf
                                       mode=700
                                       type=binary
                                       exclude=*~
                                       exclude=#*
                                       exclude=*,v
                                       purge=true
                                       server=$(policyhost)
                                       trustkey=true
                                       encrypt=true
```

You've seen this update.conf file before; we're simply getting it into place without using cf.preconf this time. The convenience of FAI's fcopy command makes cf.preconf unnecessary here.

Finally, we had to override an error from the `postfix` installation involving a missing `root` alias. In the file /srv/fai/config/hooks/savelog.LAST.source, we changed:

```
myignorepatterns="XXXXX"
```

to:

```
myignorepatterns="/etc/aliases exists, but does not have a root alias"
```

This change allows the host to fully install without having to stop for this error.

Installing Your First Debian Host

Now we're ready to boot our host *etchlamp*. We need to start a PXE boot on the host itself, which normally involves hitting the proper key on the keyboard during boot. We don't recommend setting the BIOS on your host to boot using PXE by default, at least not at a higher preference than booting from the hard disk. The last thing you want is an accidental reinstallation the next time you reboot the host! If you really prefer to boot from PXE as the first option, you can always remove the entry for the host's MAC address in `dhcpd.conf` after successful installation.

You know that PXE boot is working when you see initial output like this (this output comes from the FAI Guide; we couldn't capture this information directly from our example systems):

```
DHCP MAC ADDR: 00 04 75 74 A2 43
DHCP.../
CLIENT IP: 192.168.1.12 MASK: 255.255.255.0  DHCP IP: 192.168.1.250
GATEWAY IP: 192.168.1.254

PXELINUX 3.31 (Debian, 2007-03-09)  Copyright (C) 1994-2005 H. Peter Anvin
UNDI data segment at:   0009D740
UNDI data segment size: 3284
UNDI code segment at:   00090000
UNDI code segment size: 24C0
PXE entry point found (we hope) at 9D74:00F6
My Ip address seems to be C0A801C0 192.168.1.12
ip=192.168.1.12:192.168.1.250:192.168.1.254:255.255.255.0
```

You'll know that FAI is working when you see output on the screen like this (again, taken from the FAI Guide):

```
------------------------------------------------
        Fully Automatic Installation  -  FAI

     FAI 3.2, 21 Aug 2007  Copyright (c) 1999-2007
     Thomas Lange  <lange@informatik.uni-koeln.de>
------------------------------------------------
```

```
Calling task_confdir
Kernel parameters: initrd=initrd.img-2.6.18-5-486 ip=dhcp
root=/dev/nfs nfsroot=/srv/fai/nfsroot boot=live
FAI_FLAGS=verbose,sshd,createvt FAI_ACTION=install
BOOT_IMAGE=vmlinuz-2.6.18-5-486
```

Once you've done the imaging and reboots, you should be able to ssh into the host:

```
# ssh etchlamp -lroot
root@etchlamp's password:

This system is running linux:linux_i686_2_6_18_6_486__1_Fri_Jun_6_21_47_01_UTC_2008
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
        You are on a private campin.net system.
        Unauthorized use is prohibited, all activity is monitored.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
## SA Staff: for local-only additions here, put content in /etc/motd-local ##-

etchlamp:~# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             3.6G  390M  3.0G  12% /
tmpfs                  78M     0   78M   0% /lib/init/rw
udev                   10M   52K   10M   1% /dev
tmpfs                  78M  4.0K   78M   1% /dev/shm
etchlamp:~#
```

The host has the cfengine-configured /etc/motd, and the disk is partitioned according to our custom settings. In addition, the ps command shows that the Apache server is running. Mission accomplished!

Overall, FAI is a pleasure to work with. The directory names and scripts are self-explanatory, the class mechanism is intuitive and easy to work with, and the packages put useful starting configuration files into place. In addition, the fai-doc package includes sample configurations for the dhcpd and tftpd daemons on the system. Even for a newbie, going from no automated installation system to a fully automated mass-installation system using FAI can happen in a matter of hours.

Employing JumpStart for Solaris

JumpStart, or Custom JumpStart as it's called by Sun, is an automatic installation system for the Solaris OS. It's based on profiles, allowing a system to match installation profiles using specific criteria such as Ethernet MAC addresses or general criteria such as the system's CPU architecture. (For more information on the general JumpStart architecture, see http://docs.sun.com/app/docs/doc/817-5506/jumpstartoverview-4?a=view.)

Using JumpStart can be an entirely hands-off process, although an unattended installation might take place off CD or DVD media and use configuration files stored on the CD. In this section we configure our systems for a hands-free installation, but we'll boot from the network, as well as use profiles and install media from the network.

In getting started, we again have a chicken-and-egg problem: we need a host to configure as our JumpStart host before we can automatically image other hosts. We'll use one Solaris 10 host to handle the three network-based JumpStart roles:

1. *Boot server*: This system provides network clients with the information they need to boot and install the operating system.

2. *Profile server*: This system hosts what the JumpStart documentation calls the "JumpStart Directory." This host shares the rules file for networked installation clients. The rules file contains information on the profile to be used, as well as pre-installation and postinstallation scripts. You can also store profile information on a local floppy or optical media, if that's a better option at your site.

3. *Install server*: This system contains the Solaris disk images used to install the Solaris operating system. One install server can support many different hardware platforms and OS releases, such as SPARC and x86, plus Solaris 8 and Solaris 10.

Follow these steps to set up a new JumpStart installation host on our network:

1. Manually install a Solaris system to use as the JumpStart server.

2. Set up the installation server role.

    a. Copy the Solaris installation media to the local disk.

    b. Share the installation media via NFS.

3. Set up the profile server.

    a. Copy the sample profiles from the Solaris installation media to a new profile directory.

    b. Export the profile directory tree via NFS.

    c. Customize the profile information for your first installation client.

4. Add an installation client.

5. Boot the installation client and watch as unattended installation commences.

We picked up a Sun Enterprise 220R Server—an older PCI-bus, Sun SPARC–based server system. We installed Solaris 10 on it using CD install media, and patched it up with the latest 0508 patch bundle. We named the host *hemingway* (after the famous author), added it to the campin.net DNS, and gave it the IP address `192.168.1.237`.

We are going to image a SPARC-based system named *aurora*, with the IP address `192.168.1.248` and the Ethernet MAC address `08:00:20:8f:70:ea`. We have placed *aurora* into our DNS as well.

Setting Up the Install Server

The first thing we'll set up is the install server, which will host the Solaris installation files and packages. Here's how to set up an install directory using a Solaris 10 DVD ISO that we copied over using `scp`:

```
# mkdir /mnt/cdrom
# lofiadm -a /jumpstart/sol10-sparc.iso
# mount -o ro -F hsfs -o ro /dev/lofi/1 /mnt/cdrom
# mount -o ro -F hsfs -o ro /dev/lofi/1 /mnt/cdrom
# cd /mnt/cdrom/Solaris_10/Tools/
```

If you have a DVD drive in the system and you're using the Volume Manager to manage removable media (the default), simply change the directory to `/cdrom/cdrom0/s0/Solaris10/Tools`.

Whether using a loopback-mounted ISO or a real DVD, issue these commands to copy the DVD image to the server's hard disk:

```
# mkdir -p /jumpstart/Sol10sparc
# ./setup_install_server /jumpstart/Sol10sparc
Verifying target directory...
Calculating the required disk space for the Solaris_10 product
Calculating space required for the installation boot image
Copying the CD image to disk...
Copying Install Boot Image hierarchy...
Install Server setup complete
```

You'll need to verify that this new install directory is exported over NFS. Run the `share` command and `grep` the pathname:

```
# share | grep '/jumpstart/Sol10sparc'
-                /jumpstart/Sol10sparc   ro,anon=0    ""
```

You should see the preceding output. If not, check the `/etc/dfs/dfstab` file for an entry like this:

```
share -F nfs -o ro,anon=0 /jumpstart/Sol10sparc/
```

Add the entry it if it's missing. Once that entry is in place, verify that the NFS service is running. Issue this command on Solaris 10:

```
# svcs -l svc:/network/nfs/server:default
```

If it's not running, enable it with this command:

```
# svcadm enable svc:/network/nfs/server:default
```

Ensure the install-server directory is shared:

```
# shareall
```

If you encounter problems, see the Sun docs here: `http://docs.sun.com/app/docs/doc/817-5504/6mkv4nh3i?a=view`. The documentation is thorough, so you should be able to work out any problems.

Setting Up the Profile Server

The directory containing the `rules` file, the `rules.ok` file, and the profiles is called the JumpStart directory, and the server that hosts the JumpStart directory is called the profile server. First create the directories we'll use:

```
# mkdir /jumpstart/profiles
# mkdir /jumpstart/profiles/aurora
# cd /jumpstart/profiles/aurora
```

Next, copy over the sample profiles, which you'll need to validate the new `rules` file (they're also useful as a reference):

```
# cp -r /mnt/cdrom/Solaris_10/Misc/jumpstart_sample /jumpstart/profiles/
```

Next, share out this directory over NFS by adding this line to `/etc/dfs/dfstab`:

```
share -F nfs -o ro,anon=0 /jumpstart/profiles
```

Restart the `nfs` daemon:

```
# /usr/sbin/svcadm restart nfs/server
```

Now validate the addition:

```
# /usr/sbin/share|grep profile
-               /jumpstart/profiles   ro,anon=0    ""
```

Creating the Profile

The `profile` file is a text file that describes the software to be installed on a system. A profile describes aspects of the configuration such as the software group to install and the disk partition (slice) layout. The format is easy to understand, and because we're taking advantage of the sample configuration files included with the Solaris installation media, we can simply modify an existing profile to suit our needs.

The Sun online documentation is very good. For the complete syntax and all possible options for JumpStart profiles, please refer to `http://docs.sun.com/app/docs/doc/817-5506/preparecustom-53442?a=view`.

We'll start our profile with an example profile from the `jumpstart_sample` directory:

```
$ pwd
/jumpstart/profiles/aurora
$ cp ../jumpstart_sample/host_class .
$ mv host_class basic_prof
```

Edit the file `basic_prof` to suit your needs. We chose to install the entire Solaris 10 distribution with the package `SUNWCXall`, and we set up two filesystems and a swap slice. Here are the contents of `basic_prof`:

```
Install_type    initial_install
System_type     standalone
partitioning    explicit
filesys         c0t0d0s0 10000 /
filesys         c0t0d0s1 1024 swap
filesys         c0t0d0s7 free /opt
cluster         SUNWCXall
```

The `Install_type` keyword is required in every profile. Besides `initial_install`, other possible values for that keyword include `upgrade` and `flash_install` for upgrades and installations via a flash archive, respectively (a flash archive is a system image, not unlike a tarball snapshot of a system). The `System_type` keyword specifies that the system is to be

installed as a stand-alone system. We explicitly lay out the disk with a 10,000MB root slice and a 1,024MB swap slice, and we allocate the remaining space to the /opt filesystem.

Next, we'll test our profile. This step is optional but recommended. In place of /mnt/cdrom, give the base path to your Solaris DVD:

```
# pwd
/jumpstart/profiles/aurora
# /usr/sbin/install.d/pfinstall -D -c /mnt/cdrom/  basic_prof

Parsing profile
        1: install_type        initial_install
        2: system_type standalone
```

For this to work, you need to be on a system running the same OS version and hardware platform as the system for which you're setting up the profile. See http://docs.sun.com/app/docs/doc/817-5506/preparecustom-25808?a=view for more details.

The output of pfinstall goes on for many, many screens, but eventually should end with this:

```
Installation log location
      - /a/var/sadm/system/logs/install_log (before reboot)
      - /var/sadm/system/logs/install_log (after reboot)

Mounting remaining file systems

Installation complete

Test run complete. Exit status 0.
```

Successful completion of pfinstall means that our profile is ready.

Creating the sysidcfg File

The sysidcfg file is a preconfiguration file you use to configure a wide variety of basic system settings, including but not limited to:

- Time-zone information

- IP address and route setting

- Directory settings (e.g., DNS, Lightweight Directory Access Protocol, Network Information Service)

- Graphics and keyboard settings

- Security policy

- Language information

- Root password

The `sysidcfg` file isn't technically part of the profile (because it's not included in the `rules` file); it's used earlier than profile information in the JumpStart installation process. We do store it in the same profile directory where the rest of *aurora*'s JumpStart configuration files are kept, simply because it is convenient to do so. (For this reason, we describe it here in the section about setting up your profile server.)

Like the rest of our JumpStart files, `sysidcfg` is a text file. We created it for the host *aurora* in the `/jumpstart/profiles/aurora` directory, with these contents:

```
system_locale=en_US
terminal=vt100
name_service=DNS {domain_name=campin.net name_server=192.168.1.1
                  search=campin.net,home.campin.net}
network_interface=PRIMARY {default_route=192.168.1.1
                           netmask=255.255.255.0
                           protocol_ipv6=no}
security_policy=NONE
timezone=US/Pacific
timeserver=localhost
nfs4_domain=campin.net
root_password=FOOqi4sBARbaz
```

SYSIDCFG AND IP ADDRESS ASSIGNMENT

Note that you cannot specify the IP address of a Solaris system in the `sysidcfg` file after the system gets its IP address from Reverse Address Resolution Protocol (RARP) and the network-boot process (as we're configuring here). The installation will fail when the host tries to find a matching rule in the `rules.ok` file—you'll get an error that no matching rules were found.

We specified `timeserver=localhost` so that the installation would assume that the local time was okay. We'll configure network-based time synchronization using cfengine after initial host installation (in Chapter 7).

Experienced Solaris SAs will recognize these system settings as the earliest prompts in an interactive Solaris installation. The Custom JumpStart process uses the `sysidcfg` file to answer these questions automatically.

For more information on the sysidcfg file, see the sysidcfg(4) man page or http://docs.sun.com/app/docs/doc/817-5504/6mkv4nh2m?a=view.

Creating the postinstall Script

We need to customize our system after the JumpStart installation is complete, but before the host boots for the first time. In many JumpStart scenarios, the system doesn't boot all the way to the console login prompt, but pauses when partially done with the first boot and prompts the user for information about power management settings or the NFSv4 default domain setting. Our script works around those two issues, and also sets up cfengine when the system boots for the first time. We provide details on how to accomplish this in the following explanation of our postinstall script. We explain the script section by section:

```
#!/bin/sh

PATH=$PATH:/usr/sbin

mkdir -m 700 /a/.ssh

echo "ssh-dss AAAAB3NzaC...J5ExulczQ== nate@somehost" > /a/.ssh/authorized_keys
```

Here we put an SA's personal SSH public key into the root account's authorized_keys file. This allows for secure and easy login to the system.

■**Note** The public key placed into the root user's authorized_keys file is shortened for the purposes of this book. You can find the code samples for this chapter, including the unabbreviated version of this script, in the Downloads section of the Apress web site (http://www.apress.com).

Note that JumpStart mounts the future root filesystem at /a/. We'll use this path for the rest of this script.

The next section of code is used to detect the version of Solaris that the system is running:

```
OS_TYPE=`uname -rs`
case "$OS_TYPE" in
        "SunOS 5.10")
```

We expect to be installing only Solaris 10 systems, but it's wise to ensure that we effect changes only on the system types where we've tested this procedure. The settings for Solaris 9 would surely differ, and we don't yet know if Solaris 11 will be configured the same way. We avoid errors through defensive scripting.

Here we're creating an init script that will be run when the system is first booted:

```
cat > /a/etc/rc2.d/S99runonce <<ENDSCRIPT
```

The following procedure simply won't work from within a JumpStart installation environment, so we make it happen when the real system comes up after JumpStart. The script continues, with the contents of the S99runonce script:

```
#!/bin/sh
# used at first boot after being jumpstarted
PATH=$PATH:/usr/sbin:/opt/csw/bin:/opt/csw/sbin

LOGFILE=/var/tmp/runonce.out
LOGFILE_ERR=/var/tmp/runonce.err
exec 1>\$LOGFILE
exec 2>\$LOGFILE_ERR

# get blastwave up and running:
#  - answer "all", then "y" then "y"
pkgadd -d http://www.blastwave.org/pkg_get.pkg all <<EOM
yes
yes
EOM

cp -p /var/pkg-get/admin-fullauto /var/pkg-get/admin

pkg-get install wget gnupg textutils openssl_rt openssl_utils \
berkeleydb4 daemontools_core daemontools daemontools_core sudo cfengine
```

A software repository hosted at http://www.blastwave.org contains prepackaged freeware for Solaris systems. It resembles the popular http://www.sunfreeware.com site, but we prefer Blastwave. It is a community of capable developers and users adhering to high-quality standards for the software they upload to the site. In addition, you accomplish installation of packages from the repository through a command-line interface similar to Debian's apt-get tool. The Blastwave tool is called pkg-get. Here, upon our host's first boot, we use pkg-get to install several useful freeware tools, the most important of which is cfengine:

```
# setup cfengine key
cfkey

# bootstrap cfengine with a basic update.conf and cfagent.conf (for
# some reason we seem to need both) that will get the current configs
# from the cfengine master.

[ -d /var/cfengine/inputs ] || mkdir -p /var/cfengine/inputs
```

This next code snippet is basically our cf.preconf script from Chapter 5, integrated into the JumpStart postinstall script. In it, we set up the initial bootstrap update.conf and cfagent.conf files for the first cfagent run:

```
cat <<ENDCFCONFIG  | \
tee /var/cfengine/inputs/update.conf /var/cfengine/inputs/cfagent.conf
# created by jumpstart installation, meant to bootstrap the real
# configs from the cfengine master. If you can see this, then for some
# reason we were never able to talk to the cfengine master. :(
control:

  any::

    AllowRedefinitionOf    = ( cf_base_path workdir client_cfinput )

    # all we care about right now is the first copy
    actionsequence = (     copy.IfElapsed0 )

    domain         = ( campin.net )
    policyhost     = ( goldmaster.campin.net )

    # we host it on a Debian box
    master_cfinput = ( /var/lib/cfengine2/masterfiles/inputs )
    workdir        = ( /var/cfengine )

    #
    # Splay goes here
    #
    SplayTime      = ( 0 )
```

```
  solaris|solarisx86::
            cf_base_path    = ( /opt/csw/sbin )
            workdir         = ( /var/cfengine )

  debian::
            cf_base_path    = ( /usr/sbin )
            workdir         = ( /var/lib/cfengine2 )

  !debian.!(solaris|solarisx86)::
            # take a best guess on the path for other hosts
            cf_base_path    = ( /var/cfengine/bin )

  any::
     client_cfinput  = ( \$(workdir)/inputs )

copy:

# Everything in $(master_cfinput) on the master
# _and_ everything in its subdirectories is copied to every host.

        \$(master_cfinput)/            dest=\$(workdir)/inputs/
                                       r=inf
                                       mode=700
                                       type=binary
                                       exclude=*.lst
                                       exclude=*~
                                       exclude=#*
                                       exclude=RCS
                                       exclude=*,v
                                       purge=true
                                       server=\$(policyhost)
                                       trustkey=true
                                       encrypt=true


ENDCFCONFIG


/opt/csw/sbin/cfagent -qv

# move myself out of the way
mv /etc/rc2.d/S99runonce /etc/rc2.d/.s99runonce
ENDSCRIPT
```

The `/etc/rc2.d/S99runonce` script runs only once, and upon completion it moves itself to a file name that won't be executed by Solaris upon subsequent boots:

```
        chmod 755 /a/etc/rc2.d/S99runonce
                ;;
esac

# configure power management
sed s/unconfigured/noshutdown/ /a/etc/power.conf > /a/etc/power.conf.sed
mv /a/etc/power.conf.sed /a/etc/power.conf

# permit root login over ssh
sed 's/^PermitRootLogin no/PermitRootLogin yes/' /a/etc/ssh/sshd_config > \
 /a/etc/ssh/sshd_config.sed
mv /a/etc/ssh/sshd_config.sed /a/etc/ssh/sshd_config

# prevent prompts on first boot about power management
sed 's/^CONSOLE/\#CONSOLE/' /a/etc/default/login > /a/etc/default/login.sed
mv /a/etc/default/login.sed /a/etc/default/login

# prevent prompts on first boot about the NFS domain
touch /a/etc/.NFS4inst_state.domain

cat > /a/etc/.sysidconfig.apps <<EOSYS
/usr/sbin/sysidnfs4
/usr/sbin/sysidpm
/lib/svc/method/sshd
/usr/lib/cc-ccr/bin/eraseCCRRepository
EOSYS

cat > /a/etc/.sysIDtool.state <<EOIDT
1 # System previously configured?
1 # Bootparams succeeded?
1 # System is on a network?
1 # Extended network information gathered?
1 # Autobinder succeeded?
1 # Network has subnets?
1 # root password prompted for?
1 # locale and term prompted for?
1 # security policy in place
vt100
EOIDT
```

The rest of the entries are well commented, and shouldn't need any additional explanation. This concludes our JumpStart `postinstall` script.

Creating the rules File

The `rules` file is a text file that contains a rule for each system or group of systems on which you intend to install Solaris. Each rule uses system attributes to match a profile to the system being installed. A `rules` file entry can match a profile to a system based on the system's hostname or hardware attributes, or it can simply match all hosts to a default profile. (For more information, see `http://docs.sun.com/app/docs/doc/817-5506/preparecustom-56059?a=view`).

A `rules` file has four basic fields:

1. Rule keywords and rule value

2. Begin script

3. Profile

4. Finish script

We'll begin our `rules` file using the sample file in the `jumpstart_sample` directory:

```
$ cd /jumpstart/profiles/aurora
$ cp ../jumpstart_sample/rules .
```

Edit the `rules` file to utilize the files we've created for our first system. Here's our `rules` file, excluding comments:

```
any   - -  basic_prof finish_install.sh
```

This will match any system because of the `any` keyword. For now there's nothing system-specific in our Jumpstart setup, so having the file apply to all systems is fine. We leave the "begin script" field essentially empty by putting in a hyphen, we specify the `basic_prof` profile for the third field, and we set the "finish script" field to be our recently created `postinstall` script (documented earlier).

Now we need to validate the `rules` file, which will create the `rules.ok` file—the file actually used during installation:

```
# pwd
/jumpstart/profiles/aurora
# ../jumpstart_sample/check -r rules
Validating rules...
Validating profile basic_prof...
The custom JumpStart configuration is ok.
```

(If you encounter problems during validation, refer to the Sun documentation for troubleshooting help: `http://docs.sun.com/app/docs/doc/817-5506/` `preparecustom-11535?a=view`.) After successful validation, you now have the `rules.ok` file in the same directory as the `rules` file:

```
# pwd
/jumpstart/profiles/aurora
# ls
basic_prof        finish_install.sh  rules              rules.ok        sysidcfg
```

Adding an Installation Client

Installation clients get access to the profile-server files when you run the `add_install_cli-ent` command as `root`, which will add entries to the `/etc/bootparams` file. The settings in the `bootparams` file are handed out when clients boot using `tftp`. You don't need to take manual steps beyond the `add_install_client` step.

Our host *aurora*, whose IP address is `192.168.1.248`, is already configured in the DNS with forward and reverse entries. We collect the host's Ethernet MAC address by connecting to its serial port and watching the boot messages:

```
# cd /jumpstart/Sol10sparc/Solaris_10/Tools/
# ./add_install_client -i 192.168.1.248 -e 08:00:20:8f:70:ea -p \
hemingway:/jumpstart/profiles/aurora/ -s hemingway:/jumpstart/Sol10sparc/ -c \
hemingway:/jumpstart/profiles/aurora/ aurora sun4u
Adding Ethernet number for aurora.home.campin.net to /etc/ethers
updating /etc/bootparams
```

Now that our host *aurora* has all that it needs, we'll boot it from the network. Issue this command at the `ok` prompt:

```
{0} ok boot net - install
Resetting ...

screen not found.
Can't open input device.
Keyboard not present.  Using ttya for input and output.

Sun Ultra 2 UPA/SBus (2 X UltraSPARC-II 296MHz), No Keyboard
OpenBoot 3.11, 2048 MB memory installed, Serial #9400554.
Ethernet address 8:0:20:8f:70:ea, Host ID: 808f70ea.
```

```
Rebooting with command: boot net - install
Boot device: /sbus/SUNW,hme@e,8c00000  File and args: - install
SunOS Release 5.10 Version Generic 64-bit
Copyright 1983-2005 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
whoami: no domain name
Configuring devices.
Using RPC Bootparams for network configuration information.
Attempting to configure interface hme0...
Configured interface hme0
Beginning system identification...
Searching for configuration file(s)...
Using sysid configuration file 192.168.1.237:/jumpstart/profiles/aurora//sysidcfg
Search complete.
Discovering additional network configuration...
Completing system identification...
Starting remote procedure call (RPC) services: done.
System identification complete.
Starting Solaris installation program...
Searching for JumpStart directory...
Using rules.ok from 192.168.1.237:/jumpstart/profiles/aurora/.
Checking rules.ok file...
Using profile: basic_prof
Using finish script: finish_install.sh
Executing JumpStart preinstall phase...
```

With our carefully configured postinstallation script, the system should boot back up into Solaris without prompts at the console for information such as power management settings or the NFSv4 default domain. It's entirely possible that your list of packages, if it differs from the ones in the basic_profile profile used here, could generate interactive prompts during the first boot. If so, you'll need to take steps in either the JumpStart configuration files or the postinstallation script to configure the host properly during installation.

The host *aurora* booted up into multiuser mode (runlevel 3) without any problems, and when we first connected via SSH we were greeted with our site-specific message of the day as configured by cfengine:

```
This system is running solaris:sunos_sun4u_5_10_Generic_127127_11
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        You are on a private campin.net system.
        Unauthorized use is prohibited, all activity is monitored.
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
## SA Staff: for local-only additions here, put content in /etc/motd-local ##
```

Success!

If your system doesn't boot from the network, check your MAC and IP addresses used in the `add_install_client` command. If those settings appear to be correct, check that you have `tftpd` running under IPv4, not just IPv6. On Solaris 10, edit `/etc/inetd.conf` and make sure this line is there:

```
tftp  dgram udp   wait    root    /usr/sbin/in.tftpd   in.tftpd -s /tftpboot
```

If you have to add it, make sure that you run this code afterward to convert the `inetd.conf` entry to a proper Service Management Facility (SMF) service:

```
# /usr/sbin/inetconv -i /etc/inet/inetd.conf
```

At this point, we've set up the three Custom JumpStart roles on our single Solaris installation host, and we've imaged a new system. We realize that JumpStart has a steeper learning curve than FAI, but stick with it if you encounter problems. Once you have profiles and postinstallation scripts working to your liking, JumpStart will prove invaluable due to the unattended and consistent imaging it provides for all new Solaris hosts at your site.

Kickstart for Red Hat

The automated installation system for Red Hat Linux is called Kickstart. This system uses a single configuration file, called a *kickstart* file, to answer all the questions that would normally be asked during interactive installation.

Kickstart resembles FAI and JumpStart in that it supports network booting (PXE, in this case), followed by a fully unattended installation. One of its main strengths is that Red Hat makes available a graphical utility to create or modify kickstart files, called Kickstart Configurator. This tool helps reduce errors and explain the meaning of fields in the file. It further proves its friendliness toward the SA by displaying the raw textual content of the file for the SA's inspection (or even further modification). So Kickstart appeals to first-time users as well as seasoned veterans.

We cover Kickstart for Red Hat Enterprise Linux version 5.2. Fedora and CentOS JumpStart configuration should be similar, but we make no attempt here to cover the differences.

Red Hat has very good documentation on Kickstart in its installation guide: `http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.2/html/Installation_Guide/index.html`. We'll cover just the basics required to get Kickstart running and to install a particular host configuration.

Performing a PXE-Boot Kickstart Installation

Follow these steps to perform a Kickstart installation from the network:

1. Create the kickstart file.

2. Create and share the installation tree via NFS.

3. Place the kickstart file in the NFS share.

4. Configure TFTP for PXE booting.

5. Start the TFTP service.

6. Configure one or more hosts for network boot.

7. Configure DHCP.

8. Boot your client from the network using PXE boot and let the installation commence.

Getting the Kickstart Host

Once again, we're faced with the chicken-and-egg problem of where to get our installation host, in this case for Red Hat Linux. We chose again to use VMware, and we performed an interactive installation from DVD. Instead of covering the entire installation, we'll just mention a couple of important points:

- At the firewall screen during the installation, we chose to allow NFSv4 and SSH traffic.

- At the SELinux screen we chose to disable SELinux.

We named the system *rhmaster* and gave it the IP address 192.168.1.251.

Creating the Kickstart File

The kickstart file is a text file containing a series of keywords. Order is important in the file, which is one of the main reasons for using the graphical Kickstart Configurator application.

Every Red Hat Linux installation, whether performed interactively or via Kickstart, stores a kickstart file at /root/anaconda-ks.cfg documenting the way the system was installed. You can use this file to choose the same installation options again on many hosts, or to restore the host's OS installation in the event that it fails (assuming the file was saved in a safe place!).

The Kickstart Configurator application can open a preexisting kickstart file for editing, or start a new file from scratch. To use the application, you'll need to run a graphical desktop. Of course it's possible to display X Window System applications on a remote display, but we won't illustrate how to do that here.

We recommend starting Kickstart Configurator from a terminal window. This is because the documentation claims that the application path is `/usr/sbin/system-config-kickstart`, but on our system it is installed in `/usr/bin`. Try executing both paths inside a terminal window.

If you don't have either, install the `system-config-kickstart` package and try again.

Basic Configuration Screen

Launching Kickstart Configurator lands you at the Basic Configuration screen. Select `file` ➤ `open` ➤ `anaconda-ks.cfg` in the `root` user's home directory. Starting out your Kickstart configuration with the settings from your existing system will make this process easier. You should not specify the same installation key as your existing system. You might want to change the `root` password used for the new system. The single most important setting on this screen is probably the "Reboot system after installation" box. If you don't check this box, your system will simply pause after completion of the Kickstart installation. That's probably not what you want.
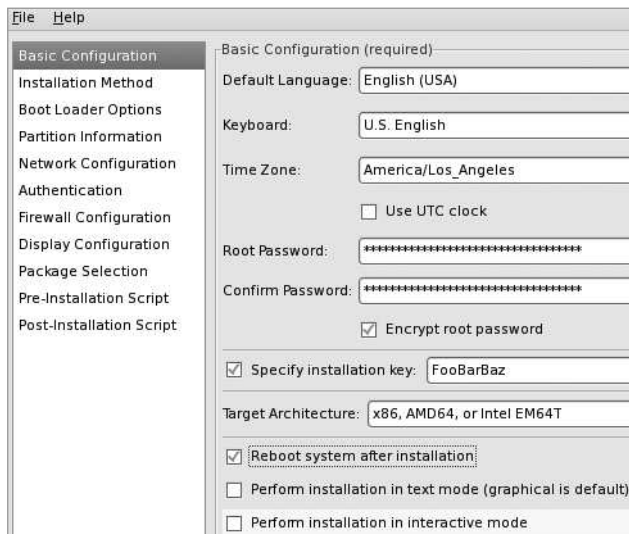
Installation Method Screen

Select the Installation Method item in the left-hand pane, keep "Perform new installa-tion" selected, and select "NFS" under "Choose the installation method." When you do that, you'll see two new text boxes that require entries: "NFS Server" and "NFS Directory." According to the Red Hat installation documentation, the latter needs to be the "directory containing the variant directory of the installation tree." In our case, we're installing the Server variant, and the Server directory we'll set up is `/kickstart/rhel5_2/Server/`. In the "NFS Server" box, enter our Kickstart server (*rhmaster*) host's IP (`192.168.1.251`), and in the "NFS Directory" box, enter the filesystem location where you plan to copy the DVD for later installation (in our case it is `/kickstart/rhel5_2`).
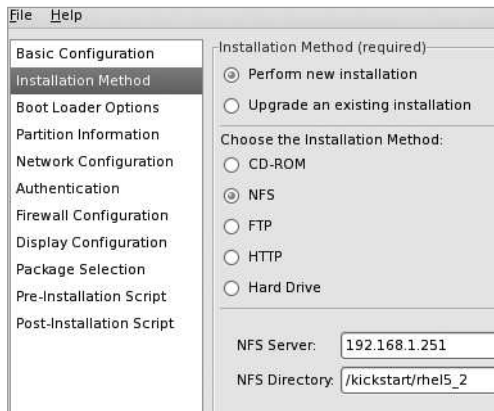
Boot Loader Options Screen

Next, select Boot Loader Options in the left-hand pane. This panel will be disabled if you selected an architecture other than x86 or x86_64. Uncheck "Use GRUB password."

As for the "Kernel parameters" field, keep the `quiet` parameter. It filters kernel mes-sages during boot to show only warning and higher-severity kernel messages. The `rhgb` parameter is a Red Hat–specific option to enable the Red Hat graphical boot facility, so we'll keep it.
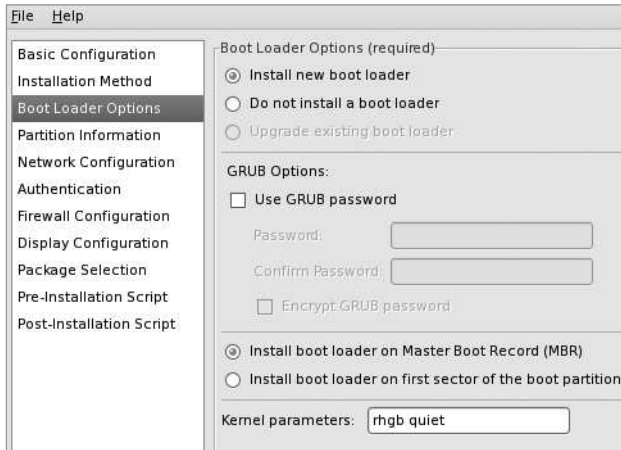
Partition Information Screen

Next, click the Partition Information entry in the left-hand pane. We're installing on new machines, so select "Clear Master Boot Record," "Remove all existing partitions," and "Initialize the disk label." It is always wise to take control of the entire disk for a server installation to fully utilize disk space and to clear the drive of any previous contents.

To add a partition, click the "Add" button. You'll get a pop-up window that lets you configure the first partition.

Configure swap first to be a fixed size, then click "OK." This will send you back to the main Partition Information screen. Once there, click "Add" again to add a root partition that fills up the rest of the disk.

Once you've configured those two partitions, your Partition Information screen will look like this:
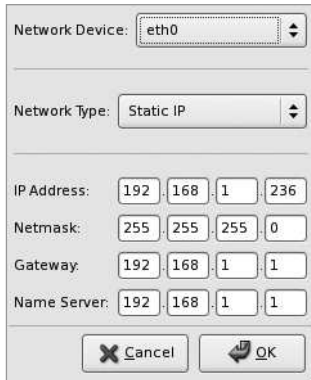
Network Configuration Screen

Select the Network Configuration screen and set up a network device. Edit your network interfaces as appropriate.
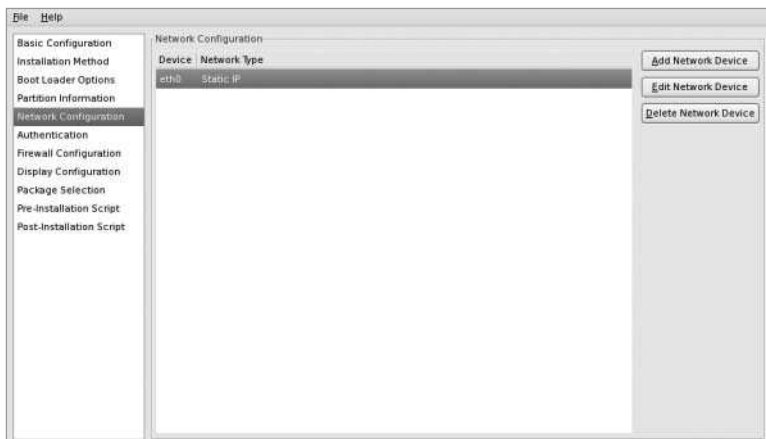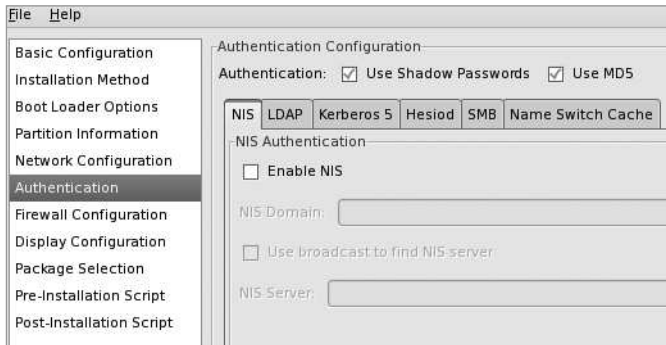
Click the "Edit Network Device" button and update the "IP Address" setting to a different static IP—the one for the new host.

Authentication Screen

Select the Authentication entry in the left-hand pane. You don't need to change any settings; "Use Shadow Passwords" and "Use MD5" should already be checked.

Firewall Configuration Screen

Select the Firewall Configuration entry in the left-hand pane. Keep the "Enable firewall" security setting and set SELinux to "Disabled." Don't set any trusted devices. Under "Trusted services," check "WWW (HTTP)" and keep SSH checked.
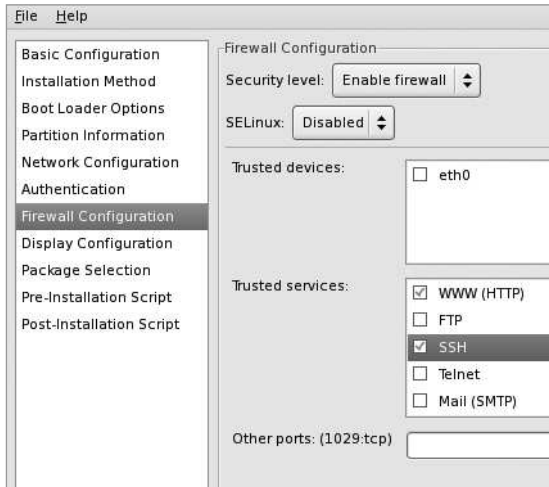
Display Configuration Screen

In the Display Configuration screen, uncheck "Configure the X Window System," which grays out the rest of the screen. You should still be able to display X apps remotely on another system if you need to, but otherwise you probably won't need X on the host.
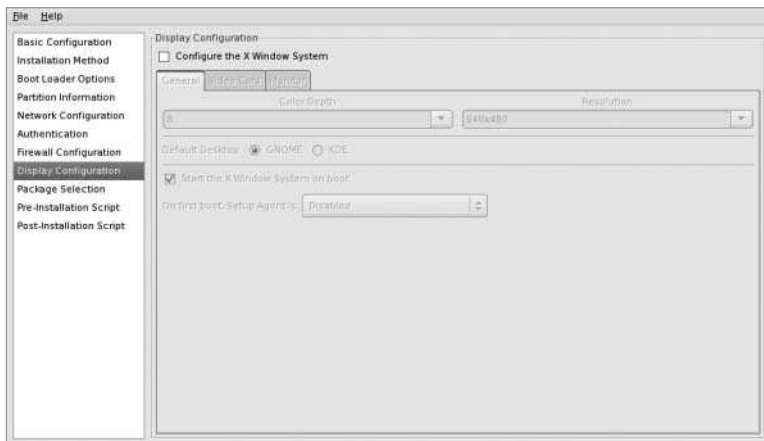
Package Selection Screen

Select the Package Selection entry in the left-hand pane. The middle pane will already be on "Desktop Environments," and it has "GNOME Desktop Environment" selected in the right-hand pane. You can uncheck it, however, because we don't need it on a server.
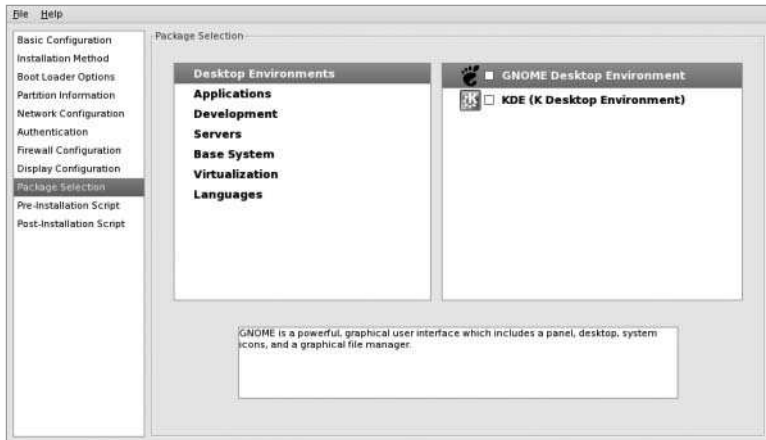
Now select "Applications" in the middle pane. In the right-hand pane, keep only "Editors" and "Text-based Internet" checked.
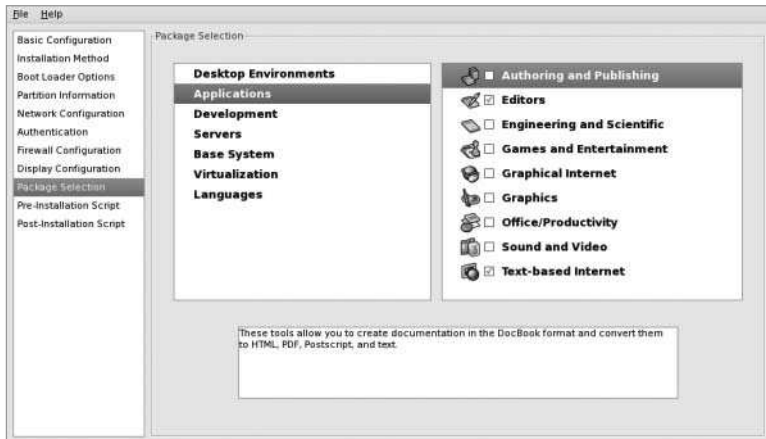
Under "Development" in the middle pane, don't select any check boxes:
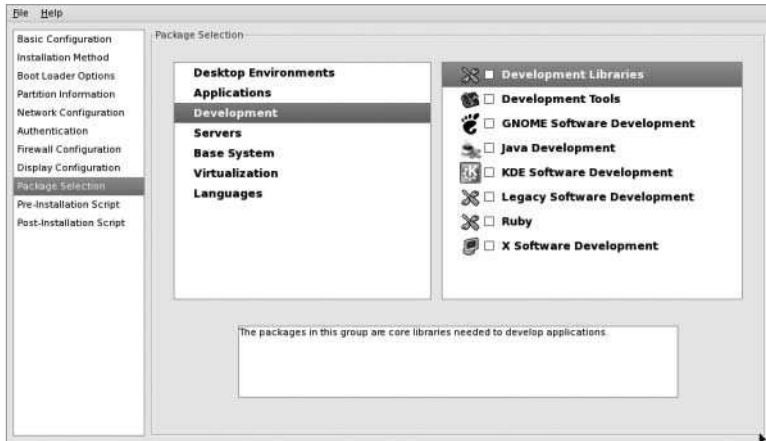
Under "Servers" in the middle pane, select only "Web Server":
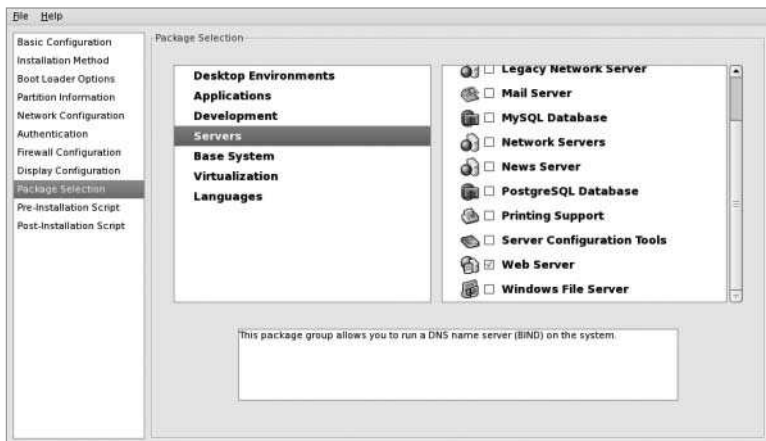
Under "Base System" in the middle pane, select "Administration Tools," "Base," and "Legacy Software Support":
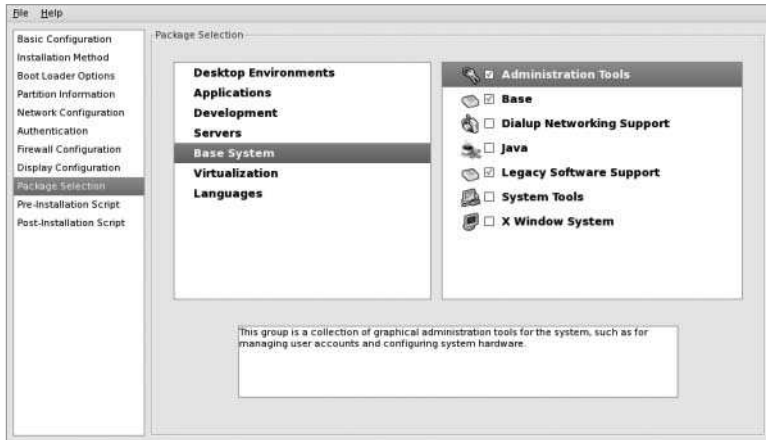
Now select "Virtualization" in the middle pane and uncheck "Virtualization" in the right-hand pane:
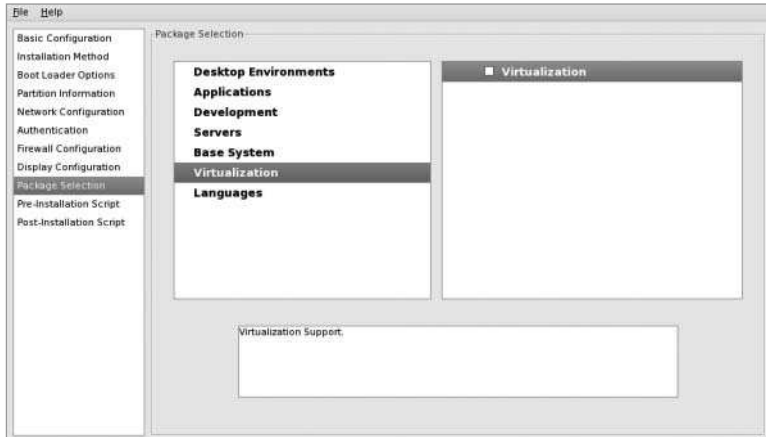
When you select "Languages" in the middle pane, you'll see that none of the languages listed in the right-hand pane are selected. Keep it that way:
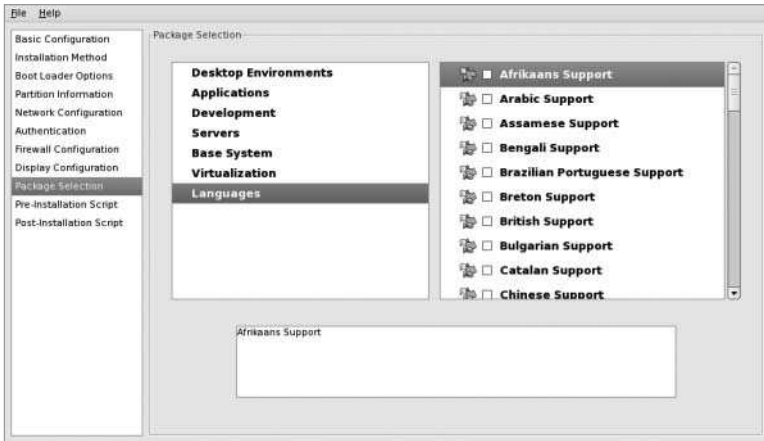
Preinstallation Script Screen

Select Pre-Installation Script in the left-hand pane and leave the screen's text box blank:
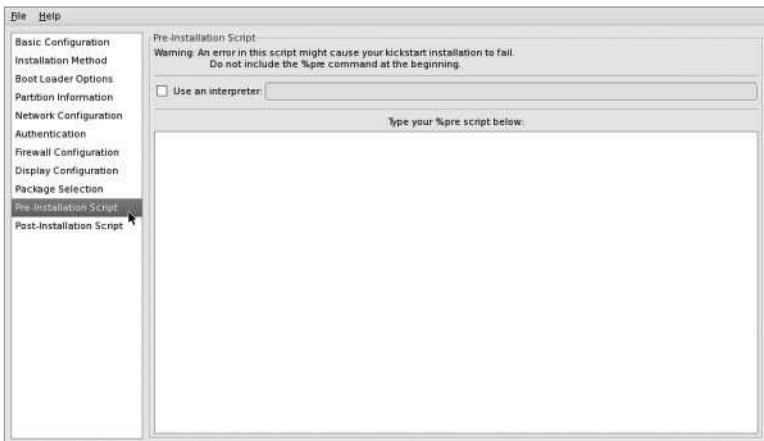
Postinstallation Script Screen

Select Post-Installation Script in the left-hand pane and paste in this small script to copy over some cfengine binaries and to run `cf.preconf` at boot:

```
[ -d /mnt/temp ] || mkdir /mnt/temp

if mount -o nolock 192.168.1.251:/kickstart /mnt/temp -s
then
        mkdir -p /usr/pkg
        cp -rp /mnt/temp/cfengine-2.2.7 /usr/pkg
        ln -s /usr/pkg/cfengine-2.2.7 /usr/pkg/cfengine

        echo "/usr/sbin/ntpdate 0.debian.pool.ntp.org 1.debian.pool.ntp.org \
2.debian.pool.ntp.org 3.debian.pool.ntp.org" >> /etc/rc.d/rc.local
        echo "/usr/pkg/cfengine/sbin/cf.preconf" >> /etc/rc.d/rc.local
        echo "/usr/pkg/cfengine/sbin/cfagent –qv" >> /etc/rc.d/rc.local

        umount /mnt/temp

else
        echo "Remote mount failed, please troubleshoot"
        exit 1
fi
```
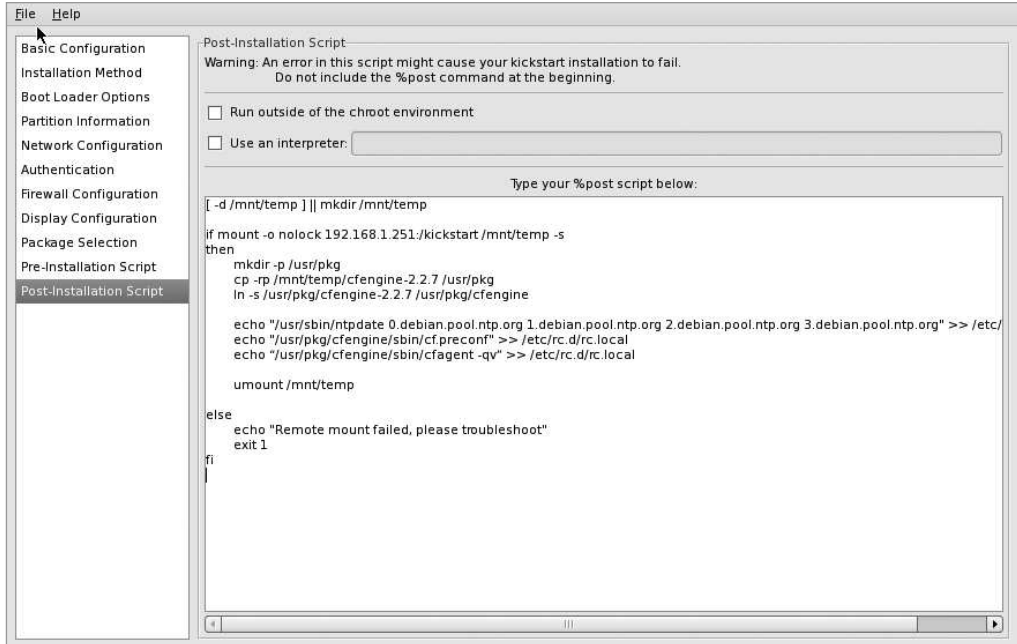

We're done! Save the file to /root/ks-192.168.1.236.cfg.

Kickstart File Contents

Here's the full `ks.cfg` file:

```
#platform=x86, AMD64, or Intel EM64T
# System authorization information
auth  --useshadow  --enablemd5
# System bootloader configuration
bootloader --append="rhgb quiet" --location=mbr --driveorder=sda
# Clear the Master Boot Record
zerombr
# Partition clearing information
clearpart --all --initlabel
# Use graphical install
graphical
# Firewall configuration
firewall --enabled --http --ssh
# Run the Setup Agent on first boot
firstboot --disable
key F00ac6b29f3e8BAR
# System keyboard
keyboard us
# System language
lang en_US
# Installation logging level
logging --level=info
# Use NFS installation media
nfs --server=192.168.1.251 --dir=/kickstart/rhel5_2
# Network information
network --bootproto=static --device=eth0 --gateway=192.168.1.1 --ip=192.168.1.236
 --nameserver=192.168.1.1 --netmask=255.255.255.0 --onboot=on
# Reboot after installation
reboot
#Root password
rootpw --iscrypted F00N772Cl.o$Y1TP4qlObokg.VRikneBAR

# SELinux configuration
selinux --disabled
# Do not configure the X Window System
skipx
# System timezone
timezone --isUtc America/Los_Angeles
```

```
# Install OS instead of upgrade
install
# Disk partitioning information
part swap --bytes-per-inode=4096 --fstype="swap" --size=768
part / --bytes-per-inode=4096 --fstype="ext3" --grow --size=1
%post
[ -d /mnt/temp ] || mkdir /mnt/temp

if mount -o nolock 192.168.1.251:/kickstart /mnt/temp -s
then
        mkdir -p /usr/pkg
        cp -rp /mnt/temp/cfengine-2.2.7 /usr/pkg
        ln -s /usr/pkg/cfengine-2.2.7 /usr/pkg/cfengine

        echo "/usr/sbin/ntpdate 0.debian.pool.ntp.org 1.debian.pool.ntp.org \
2.debian.pool.ntp.org 3.debian.pool.ntp.org" >> /etc/rc.d/rc.local
        echo "/usr/pkg/cfengine/sbin/cf.preconf" >> /etc/rc.d/rc.local
        echo "/usr/pkg/cfengine/sbin/cfagent –qv" >> /etc/rc.d/rc.local

        umount /mnt/temp

else
        echo "Remote mount failed, please troubleshoot"
        exit 1
fi


%packages
@base
@core
@web-server
@admin-tools
@text-internet
@legacy-software-support
@editors
emacs
kexec-tools
bridge-utils
device-mapper-multipath
xorg-x11-utils
xorg-x11-server-Xnest
libsane-hpaio
-sysreport
```

Creating the Installation Tree and Making It Available

We'll use a Red Hat Enterprise Linux 5.2 DVD ISO to create our installation tree. First, mount the DVD as a loopback filesystem:

```
# mount -t iso9660 /root/rhel52.iso /mnt/iso/ -o loop
```

Use the df command to verify that it is mounted properly:

```
# df -h
Filesystem              Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup00-LogVol00
                        66G   8.0G   55G  13% /
/dev/sda1               99M   21M   74M  22% /boot
tmpfs                   61M    0    61M   0% /dev/shm
/root/rhel52.iso        2.9G  2.9G    0  100% /mnt/iso
```

Now you can create the installation tree directory:

```
# mkdir -p /kickstart/rhel5_2
# cp -Rp /mnt/iso/* /kickstart/rhel5_2/
```

Next, we need to set up the NFS server. Navigate to System ➤ Administration ➤ Server Settings to configure NFS:

Use the system-config-nfs applet (found in the graphical desktop at system ➤ administration ➤ services) to share the /kickstart/rhel5_2 directory over NFS. Allow read-only access to the 192.168.1.0/24 subnet:
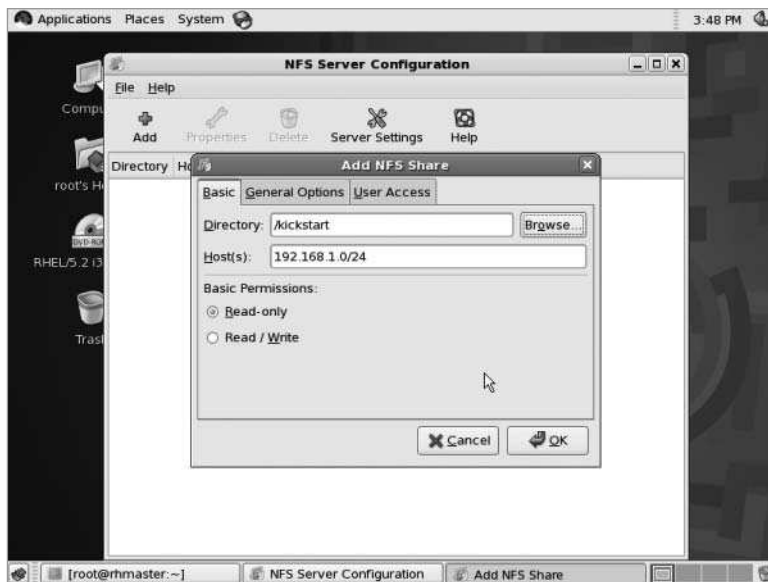
Copy the previously created kickstart file to our new NFS share.

```
# cp /root/ks-192.168.1.236.cfg /kickstart/rhel5_2/ks.cfg
```

This is the location we'll reference in the PXE boot configuration, described in the next section ("Setting Up Network Boot").

To install cfengine on your Red Hat systems, compile cfengine 2.2.7 and install it to /usr/pkg/cfengine-2.2.7 on the *rhmaster* machine. Copy the installation to /kickstart/cfengine-2.2.7 so that Kickstart clients can mount and copy the files. Then place cf.preconf in /usr/pkg/cfengine/sbin/ so that it can be copied over with the rest of the installation.

The Kickstart cf.preconf file is the same file from /var/lib/cfengine2/masterfiles/PROD/inputs/cf.preconf on the cfengine master. It is already written to bootstrap Red Hat systems, so our postinstall script simply needs to copy the cfengine binary directory to the correct location on the local system, and run cf.preconf upon boot. The postinstall script takes care of all of this.

Setting Up Network Boot

Now that we have our kickstart file ready, we need to set up network booting.

Trivial File Transfer Protocol (TFTP)

We'll need the `tftp-server` and `syslinux` packages, which aren't installed by default, according to the Red Hat Installation Guide. Use `yum` to install the packages.

Interestingly, our *rhmaster* system did already have the `tftp-server` package installed. It had even placed the files required for boot into `/tftpboot`:

```
# ls /tftpboot/linux-install/
msgs  pxelinux.0  pxelinux.cfg
```

That saves some steps. If the packages aren't on your system, here's how to populate it: the `tftp-server` package creates the `/tftpboot` directory. We'll need to create the `/tftpboot/linux-install` directory:

```
# mkdir /tftpboot/linux-install
# cp /usr/lib/syslinux/pxelinux.0 /tftpboot/linux-install
```

Create `/tftpboot/linux-install/msgs`:

```
# mkdir /tftpboot/linux-install/msgs
```

Copy the .msg files from the `isolinux/` directory on the installation tree:

```
# cp /kickstart/rhel5_2/isolinux/*msg /tftpboot/linux-install/msgs/
```

Now it is time to set up support for the release and variant of Red Hat we're planning to use. We can support different variants (server vs. workstation) and versions (Red Hat 5.1 vs. Red Hat 5.2) from the same PXE server. On our system, we'll be setting up only 5.2 Server, although we can extend it later if we need to.

We need to set up an OS-specific directory underneath `linux-install`:

```
# mkdir /tftpboot/linux-install/rhel5_2
```

Copy the `initrd.img` and `vmlinuz` files from the `images/pxeboot` directory of your installation tree to the OS-specific `tftp` directory:

```
# cp /kickstart/rhel5_2/images/pxeboot/initrd.img /tftpboot/linux-install/rhel5_2/
# cp /kickstart/rhel5_2/images/pxeboot/vmlinuz /tftpboot/linux-install/rhel5_2/
```

Next, we need to set up PXE config files. Create the `/tftpboot/linux-install/pxelinux.cfg` directory:

```
# mkdir /tftpboot/linux-install/pxelinux.cfg
```

The `/tftpboot/linux-install/pxelinux.cfg/` directory will need a file for each system to be installed. The file's name is either the hostname or IP address of the system to be booted/installed. If no matching file is found (based on IP or hostname), the config file named `default` is used. This is standard `syslinux` PXE/TFTP server configuration, and is not Red Hat–specific.

The PXE config file for our system with MAC address `00:0c:29:d1:19:82` will be `/tftpboot/linux-install/pxelinux.cfg/01-00-0c-29-d1-19-82`, and the file contents are:

```
default 1
timeout 100
prompt 1
display msgs/boot.msg
F1 msgs/boot.msg
F2 msgs/general.msg
F3 msgs/expert.msg
F4 msgs/param.msg
F5 msgs/rescue.msg
F7 msgs/snake.msg

label 1
 kernel rhel5_2/vmlinuz
 append initrd=rhel5_2/initrd.img ramdisk_size=6878 ip=dhcp \
 ks=nfs:192.168.1.251:/kickstart/rhel5_2/ks.cfg
```

Next, enable `tftp` and `xinetd`, the latter of which starts the `tftp` daemon upon connections from clients:

```
# chkconfig --level 345 xinetd on
# chkconfig --level 345 tftp on
```
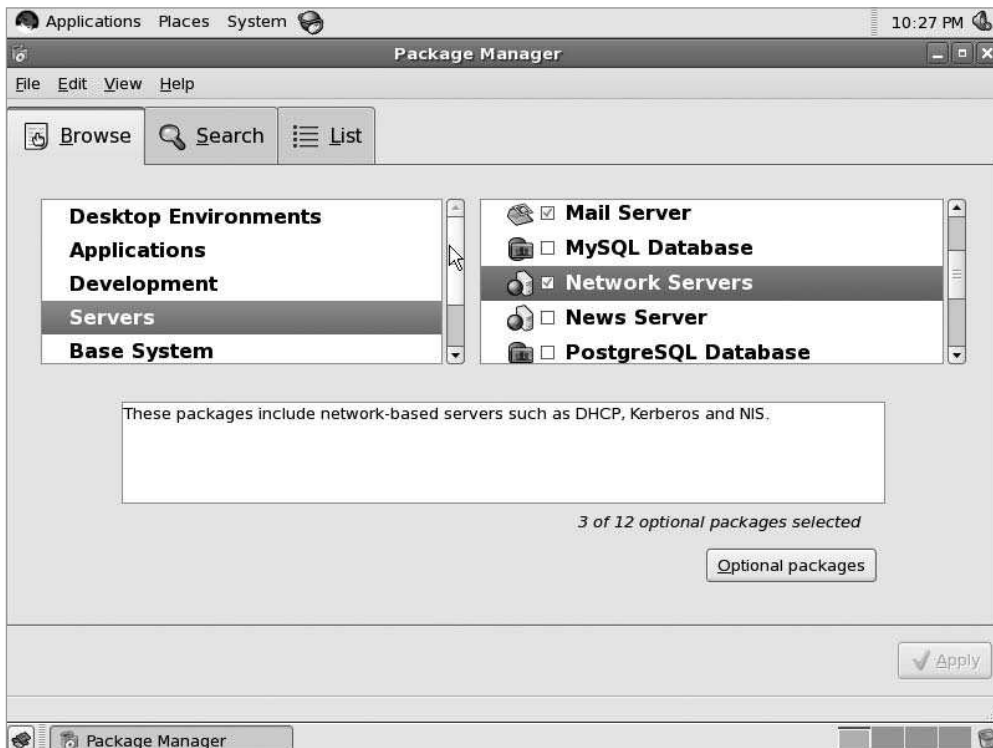
If `xinetd` was already running, restart it:

```
# /etc/init.d/xinetd restart
```

DHCP

If your Kickstart server doesn't already have DHCP installed and running, open the Add/Remove Software menu item on the Applications menu:

Under the Browse tab, select "Servers" in the left-hand pane, then "Network Servers" in the right-hand pane. Now click the "Optional packages" button:

Then click the check box for "dhcp":

Click the "Close" button, then the "Apply" button. Then select "Continue" under the "Package Selections" dialog box that pops up.

Here is the /etc/dhcpd.conf file from our Kickstart server:

```
deny unknown-clients;
option dhcp-max-message-size 2048;
use-host-decl-names on;

subnet 192.168.1.0 netmask 255.255.255.0 {
   option routers 192.168.1.1;
   option domain-name "campin.net";
   option domain-name-servers 192.168.1.1;
}

# based on error message we got without it:
ddns-update-style ad-hoc;
```

```
allow booting;
allow bootp;
class "pxeclients" {
        match if substring(option vendor-class-identifier, 0, 9) = "PXEClient";
        next-server 192.168.1.251;
        filename "linux-install/pxelinux.0";
}

host rhlamp {hardware ethernet 00:0c:29:d1:19:82;fixed-address rhlamp;}
```

We set up the host *rhlamp* as our first installation client. We gathered its MAC address during an attempted PXE boot and put the *rhlamp.campin.net* forward and reverse entries into the DNS.

Installing a Host Using Kickstart

Set the BIOS on your installation client to boot from the network first, or press whatever key is necessary to interrupt the normal boot sequence and boot using PXE.

It will be immediately apparent if the Kickstart configuration is fully functional. If the host boots properly using PXE but can't find or otherwise get all the information it needs from the ks.cfg file, it will go into an interactive installation. If it doesn't boot at all, then you need to troubleshoot your DHCP/tftpd configuration.

When it reboots after Kickstart completion, it will not be registered with RHN, which is required to use the Red Hat software channels. The tool /usr/sbin/rhnreg_ks was designed to register hosts noninteractively, such as from Kickstart installation postinstall scripts. You'll find an example in the Red Hat Installation Guide: http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.2/html/Installation_Guide/s2-kickstart2-post-examples.html.

The Proper Foundation

Our site now has the two most critical pieces of core infrastructure:

1. Automated installation

2. Automated configuration

We have the ability to deploy new Red Hat, Debian, and Solaris systems rapidly. These systems will join our infrastructure and will be automatically managed by cfengine. This puts us in the enviable position of not needing to manually log into any systems to make changes. We'll make changes centrally, and allow changes to take place via automated means—and automated means *only*.

In the next chapter, we'll take advantage of this foundation to start configuring important infrastructure services, with almost all of our activity actually taking place on the cfengine master instead of on the hosts running those services.

Automating a New System Infrastructure

Every UNIX-based site requires a similar list of infrastructure services in order to function. All sites need to keep the correct time, route e-mail from system processes (such as cron jobs, and in our case `cfexecd`) to the correct place, convert hostnames into IP addresses, and control user accounts.

We think it's only fair to warn you that this chapter won't go into great detail on the protocols and server software that we'll configure. If we had to explain DNS, NTP, SMTP, NFS, the automounter, and UNIX authentication files in great detail, the chapter would never end. Additionally, it would draw focus away from our goal of automating a new infrastructure using cfengine. We'll recommend other sources of information for the protocols and server software as we progress though the chapter.

When we refer to files in the cfengine `masterfiles` repository on our central host (*goldmaster*), we'll use paths relative to `/var/lib/cfengine2/masterfiles`. This means that the full path to `PROD/inputs/tasks/os/cf.ntp` is `/var/lib/cfengine2/masterfiles/PROD/inputs/tasks/os/cf.ntp`.

Implementing Time Synchronization

Many programs and network protocols fail to function properly when the clock on two systems differ by more than a small amount.

The lack of time synchronization can cause extensive problems at a site. These are the most common:

- E-mail messages have the incorrect time.

- Log entries cannot be correlated across different systems.

- Monitoring alerts specify the incorrect time for outages.

- Authentication transactions fail.

- Automation-system changes based on file-modification times work improperly.

- Software build tools such as *make* (which depend on file-modification times) break.

We'll tackle Network Time Protocol (NTP) configuration before any other infrastructure setup tasks. We won't go into the level of detail that you'll want if you're deploying NTP across hundreds or thousands of systems. If that's the case, accept our apologies and proceed over to `http://www.ntp.org` to browse the online documentation, or head to your nearest bookseller and pick up a copy of *Expert Network Time Protocol* by Peter Rybaczyk (Apress, 2005).

The fact that we already have six hosts at our example site without synchronized clocks is a potential problem. The `cfservd` daemon will refuse to serve files to clients if the clocks on the systems differ by more than one hour. You can turn off this behavior with this setting in `cfservd.conf`:

```
DenyBadClocks off
```

It might make sense to turn it off during the initial bootstrapping phase at your site, before you deploy NTP.

NTP is the Internet standard for time synchronization. Interestingly, it's one of the oldest Internet standards still in widespread use. NTP is a mechanism for transmitting the universal time (UTC, or Coordinated Universal Time) between systems on a network. It is up to the local system to determine the local time zone and Daylight Saving settings, if applicable. NTP has built-in algorithms for dealing with variable network latency, and can achieve rather impressive accuracy even over the public Internet.

External NTP Synchronization

The ntp.org web site has a list of public NTP servers here: `http://support.ntp.org/bin/view/Servers/WebHome`. These are groups of public NTP servers that use round-robin DNS to enable clients to make a random selection from the group. Both Red Hat and Debian have NTP pools set up this way, and the NTP packages from those distributions utilize these pools by default.

Our intention is to have two of our internal servers synchronize to an external source, and have the rest of our systems synchronize from those two. This is the polite way to utilize a public NTP source: placing as little load as possible on it. We don't want a single system to perform off-site synchronization for our entire network because it becomes a single point of failure. We generally want to set up DNS aliases for system roles such as NTP service, but NTP configuration files use IP addresses. This actually works out well because we have yet to set up internal DNS.

Internal NTP Masters

We'll use our cfengine master host (*goldmaster.campin.net*) and our Red Hat Kickstart system (*rhmaster.campin.net*) as the two systems that sync to an external NTP source.

■**Note** There is no reason to choose Linux over Solaris systems to handle this role. You should find it quite easy to modify this procedure to use one or more Solaris systems to synchronize off site instead, and have all other systems synchronize to the internal Solaris NTP servers.

The Red Hat system already had `ntpd` installed (the `ntp` RPM package). If you wish to graphically configure NTP on Red Hat, you'll need to have the `system-config-date` RPM installed. Basic NTP configuration is straightforward, so we'll stick with text-based methods of configuration.

The Debian system didn't have the required packages installed, so we used `apt-get` to install the `ntp` package. We went back to our FAI configuration and added the line `ntp` to the file `/srv/fai/config/package_config/FAIBASE` so that all future Debian installs have the package by default. Our Kickstart installation process already installs the `ntp` RPM, so we don't have to make any Kickstart modifications.

Here is the `/etc/ntpd.conf` file that we'll use on our systems that synchronize to off-site NTP sources:

```
driftfile /var/lib/ntp/ntp.drift
statsdir /var/log/ntpstats/

statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

# pool.ntp.org maps to more than 300 low-stratum NTP servers.
# Your server will pick a different set every time it starts up.
server 0.debian.pool.ntp.org iburst
server 1.debian.pool.ntp.org iburst
server 2.debian.pool.ntp.org iburst
server 3.debian.pool.ntp.org iburst
```

```
# By default, exchange time with everybody, but don't allow configuration.
# See /usr/share/doc/ntp-doc/html/accopt.html for details.
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery

# Local users may interrogate the ntp server more closely.
restrict 127.0.0.1
restrict ::1

# allow the local subnet to query us
restrict 192.168.1.0 mask 255.255.255.0 nomodify notrap
```

Both Red Hat and Debian have a dedicated user to run the NTP daemon process. The user account, named "ntp," will need write access to the /var/lib/ntp directory.

When you name a subnet using the restrict keyword and omit the noquery keyword, the server allows NTP client connections from that subnet.

Configuring the NTP Clients

Now that we have working NTP servers on our network, we need configuration files for the Linux (both Red Hat and Debian) and Solaris systems on our network. We refer to the systems running NTP to synchronize only with internal hosts as NTP "clients."

Solaris 10 NTP Client

You'll find it easy to configure a single Solaris 10 system to synchronize its time using NTP. We will automate the configuration across all our Solaris systems later, but will first test our configuration on a single host to validate it. Simply copy /etc/inet/ntp.servers to /etc/inet/ntp.conf, and comment out these lines:

```
server 127.127.XType.0
fudge 127.127.XType.0 stratum 0
keys /etc/inet/ntp.keys
trustedkey 0
requestkey 0
controlkey 0
```

Add lines for our internal NTP servers:

```
server 192.168.1.249
server 192.168.1.251
```

Create the file /var/ntp/ntp.drift as root using the touch command, and enable the ntp service:

```
# touch /var/ntp/ntp.drift
# /usr/sbin/svcadm enable svc:/network/ntp
```

It's really that easy. Check the /var/log/messages log file for lines like this, indicating success:

```
Jul 27 18:05:30 aurora ntpdate[995]: [ID 558275 daemon.notice] adjust time server
192.168.1.249 offset 0.008578 sec
```

Red Hat and Debian NTP Client

We use the same NTP configuration-file contents for all the remaining Debian and Red Hat hosts at our site, shown here:

```
driftfile /var/lib/ntp/ntp.drift
statsdir /var/log/ntpstats/

statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable

# By default, exchange time with everybody, but don't allow configuration.
# See /usr/share/doc/ntp-doc/html/accopt.html for details.
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery

# Local users may interrogate the ntp server more closely.
restrict 127.0.0.1
restrict ::1

restrict 192.168.1.249    nomodify        # goldmaster.campin.net
restrict 192.168.1.251    nomodify        # rhmaster.campin.net
```

You'll notice that these file contents resemble the contents of the configuration file used on the hosts that sync off site. The difference here is that we have no server lines, and we added new restrict lines specifying our local NTP server systems.

Copying the Configuration Files with cfengine

Now we will distribute the NTP configuration file using cfengine, including automatic `ntp` daemon restarts when the configuration file is updated. First, put the files into a suitable place in the cfengine master repository (on the host *goldmaster*):

```
# cd /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/ntp/
# ls -1
ntp.conf
ntp.conf-master
ntp.server
```

You might remember that we created the `ntp` directory back when we first set up the `masterfiles` repository. The `ntp.conf-masters` file is meant for *rhmaster* and *goldmaster*, the hosts that synchronize NTP using off-site sources. The `ntp.conf` file is for all remaining Linux hosts, and `ntp.server` is our Solaris 10 NTP configuration file.

We'll create a task file at the location `PROD/inputs/tasks/os/cf.ntp` on the cfengine master (*goldmaster*). Once the task is written, we'll import it into the `PROD/inputs/hostgroups/cf.any` file for inclusion across our entire site. Here is the task file:

```
classes:  # synonym groups:

        ntp_servers                 = (     rhmaster
                                            goldmaster
                                            )
```

Now we define a simple group of two hosts, the machines that sync off site:

```
control:

        any::
                AddInstallable          = ( restartntpd )
                AllowRedefinitionOf     = ( ntp_conf_source )

                #
                # The default ntp.conf doesn't sync off-site
                #
                ntp_conf_source = ( "ntp.conf" )

        linux::
                ntp_user        = ( "ntp" )
                ntp_conf_dest   = ( "/etc/ntp.conf" )
                drift_file      = ( "/var/lib/ntp/ntp.drift" )
```

```
        solaris|solarisx86::
                ntp_user        = ( "root" )
                ntp_conf_dest   = ( "/etc/inet/ntp.conf" )
                drift_file      = ( "/var/ntp/ntp.drift" )
                ntp_conf_source = ( "ntp.server" )

        ntp_servers::
                # the ntp.conf for these hosts causes ntpd to sync
                # off-site, and share the information with the local net
                ntp_conf_source = ( "ntp.conf-master" )
```

In the `control` section, you define class-specific variables for use in the `files` and `copy` actions:

```
files:
        # ensure that the drift file exists and is
        # owned and writable by the correct user
        any::
                $(drift_file) mode=0644 action=touch
                  owner=$(ntp_user) group=$(ntp_user)
```

If we didn't use variables for the location of the NTP drift file and the owner of the `ntpd` process, we would have to write multiple `files` stanzas. When the entry is duplicated with a small change made for the second class of systems, you face a greater risk of making errors when both entries have to be updated later. We avoid such duplication.

We also manage to write only a single `copy` stanza, again through the use of variables:

```
copy:

        any::
                $(master_etc)/ntp/$(ntp_conf_source)
                                dest=$(ntp_conf_dest)
                                mode=644
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                owner=root
                                group=root
                                define=restartntpd
```

Here we copy out the applicable NTP configuration file to the correct location for each operating system. When the file is successfully copied, the `restartntpd` class is defined. This triggers actions in the following `shellcommands` section:

```
shellcommands:

        # restart ntpd when the restartntpd class is defined
        debian.restartntpd::
                "/etc/init.d/ntp restart" timeout=30 inform=true

        # restart ntpd when the restartntpd class is defined
        redhat.restartntpd::
                "/etc/init.d/ntpd restart" timeout=30 inform=true

        # restart ntpd when the restartntpd class is defined
        (solarisx86|solaris).restartntpd::
                "/usr/sbin/svcadm restart svc:/network/ntp" timeout=30 inform=true
```

When the `ntp.conf` file is updated, the class `restartntpd` is defined, and it causes the
`ntp` daemon process to restart. Based on the classes a system matches, the `restartntpd`
class causes cfengine to take the appropriate restart action.

Note that we have two almost identical restart commands for the `debian` and `redhat`
classes. We could have reduced that to a single stanza, as we did for the `files` and `copy`
actions. Combining those into one `shellcommands` action is left as an exercise for the
reader.

Now let's look at the `processes` section:

```
processes:

        # start ntpd when it's not running
        debian::
                "ntpd" restart "/etc/init.d/ntp start"

        # start ntpd when it's not running
        redhat::
                "ntpd" restart "/etc/init.d/ntpd start"

        # this is for when it's not even enabled
        solarisx86|solaris::
                "ntpd" restart "/usr/sbin/svcadm enable svc:/network/ntp"
```

In this section, we could have used the `restartntpd` classes to trigger the delivery of
a HUP signal to the running `ntpd` process. We don't do that because a HUP signal causes
the `ntpd` process to die. For this reason, we use the init scripts on Linux and the SMF on
Solaris.

THE SOLARIS SERVICE MANAGEMENT FACILITY

The Service Management Facility, or SMF, is a feature introduced in Solaris 10 that drastically changed the way that services are started. We consider it a huge step forward in Solaris, because it allows services to start in parallel by default. Plus, through the use of service dependencies, the SMF will start services only when the services that they depend on have been properly started.

Most of the services that Solaris traditionally started using scripts in run-level directories (e.g., `/etc/rc2.d/`) are now started by the SMF. The SMF adds several other improvements over simple startup scripts:

- Services that exit will be restarted automatically several times. After a limit is reached, the SMF performs no further restarts and the service enters a "maintenance" state.

- You can use command-line utilities to query the state of any SMF-managed service, including the reason why a service failed to start.

- You'll experience faster bootup time.

- The SMF takes snapshots of service configurations automatically, making service restoration easier when errors are introduced.

To learn more about the SMF, read Sun's BigAdmin introduction here: `http://www.sun.com/bigadmin/content/selfheal/smf-quickstart.jsp`.

This task represents how we'll write many of our future cfengine tasks. We'll define variables to handle different configuration files for different system types, then use actions that utilize those variables.

The required entry in `PROD/inputs/hostgroups/cf.any` to get all our hosts to import the task is the file path relative to the `inputs` directory:

```
import:
      any::
              tasks/os/cf.motd
              tasks/os/cf.cfengine_cron_entries
              tasks/os/cf.ntp
```

If you decide that more hosts should synchronize off site, you'd simply configure an additional Linux host to copy the `ntp.conf-masters` file instead of the `ntp.conf` file. You'd need to write a slightly modified Solaris `ntp.server` config file if you choose to have a Solaris host function in this role. We haven't done so in this book—not because Solaris isn't suited for the task, but because we needed only two hosts in this role. You'd then

add a new `restrict` line to the NTP client configuration file on Linux, or a new `server` line for Solaris NTP clients. That's three easy steps to make our site utilize an additional local NTP server.

An Alternate Approach to Time Synchronization

We can perform time synchronization at our site using a much simpler procedure than running the NTP infrastructure previously described. We can simply utilize the `ntpdate` utility to perform one-time clock synchronization against a remote NTP source. To manually use `ntpdate` once, run this at the command line as `root`:

```
# /usr/sbin/ntpdate 0.debian.pool.ntp.org
20 Sep 17:09:15 ntpdate[181]: adjust time server 208.113.193.10 offset -0.00311 sec
```

Note that `ntpdate` will fail if a local `ntpd` process is running, due to contention for the local NTP TCP/IP port (UDP/123). Temporarily stop any running `ntpd` processes if you want to test out `ntpdate`.

We consider this method of time sychronization to be useful only on a temporary basis. The reason for this is that `ntpdate` will immediately force the local time to be identical to the remote NTP source's time. This can (and often does) result in a major change to the local system's time, basically a jump forward or backward in the system's clock.

By contrast, when `ntpd` sees a gap between the local system's time and the remote time source(s), it will gradually decrease the difference between the two times until they match. We prefer the approach that `ntpd` uses because any logs, e-mail, or other information sources where the time is important won't contain misleading times around and during the clock jump.

Because we discourage the use of `ntpdate`, we won't demonstrate how to automate its usage. That said, if you decide to use `ntpdate` at your site, you could easily run it from cron or a cfengine `shellcommands` section on a regular basis.

Incorporating DNS

The Domain Name System (DNS) is a globally distributed database containing domain names and associated information. Calling it a "name-to-IP-address mapping service" is overly simplistic, although it's often described that way. It also contains the list of mail servers for a domain as well as their relative priority, among other things. We don't go into great detail on how the DNS works or the finer details of DNS server administration, but you can get more information from *DNS and BIND, Fifth Edition* by Cricket Liu and Paul Albitz (O'Reilly Media Inc., 2006), and the Wikipedia entry at `http://en.wikipedia.org/wiki/Domain_Name_System`.

Choosing a DNS Architecture

Standard practice with DNS is to make only certain hostnames visible to the general public. This means that we wouldn't make records such as those for *goldmaster.campin.net* available to systems that aren't on our private network. When we need mail to route to us from other sites properly or get our web site up and running, we'll publish MX records (used to map a name to a list of mail exchangers, along with relative preference) and an A record (used to map a name to an IPv4 address) for our web site in the public DNS.

This sort of setup is usually called a "split horizon," or simply "split" DNS. We have the internal hostnames for the hosts we've already set up (*goldmaster*, *etchlamp*, *rhmaster*, *rhlamp*, *hemingway*, and *aurora*) loaded into our *campin.net* domain with a DNS-hosting company. We'll want to remove those records at some point because they reference private IP addresses. They're of no use to anyone outside our local network and therefore should be visible only on our internal network. We'll enable this record removal by setting up a new private DNS configuration and moving the private records into it.

Right about now you're thinking "*Wait*! You've been telling your installation clients to use 192.168.1.1 for both DNS and as a default gateway. What gives? Where did that host or device come from?" Good, that was observant of you. When we mentioned that this book doesn't cover the network-device administration in our example environment, we meant our single existing piece of network infrastructure: a Cisco router at 192.168.1.1 that handles routing, Network Address Translation (NAT), and DNS-caching services. After we get DNS up and running on one or more of our UNIX systems, we'll have cfengine configure the rest of our systems to start using our new DNS server(s) instead.

Setting Up Private DNS

We'll configure an internal DNS service that is utilized only from internal hosts. This will be an entirely stand-alone DNS infrastructure not linked in any way to the public DNS for *campin.net*.

This architecture choice means we need to synchronize any public records (currently hosted with a DNS-hosting company) to the private DNS infrastructure. We currently have only mail (MX) records and the hostnames for our web site (*http://www.campin.net* and *campin.net*) hosted in the public DNS. Keeping this short list of records synchronized isn't going to be difficult or time-consuming.

We'll use Berkeley Internet Name Domain (BIND) to handle our internal DNS needs.

■**Note** Be sure that the BIND software you install is resistant to the DNS protocol flaw made public in July 2008. Also, if your DNS servers are behind NAT, make sure your NAT device doesn't defeat the port randomization that works around the flaw. For more information, see the CERT advisory here: http://www. kb.cert.org/vuls/id/800113.

BIND Configuration

We'll use the *etchlamp* system that was installed via FAI as our internal DNS server. Once it's working there, we can easily deploy a second system just like it using FAI and cfengine.

First, we need to install the `bind9` package, as well as add it to the set of packages that FAI installs on the `WEB` class.

In order to install the `bind9` package without having to reinstall using FAI, run this command as the `root` user on the system *etchlamp*:

```
# apt-get update && apt-get install bind9
```

The `bind9` package depends on other packages such as `bind-doc` (and several more), but `apt-get` will resolve the dependencies and install everything required. Because FAI uses `apt-get`, it will work the same way, so we can just add the line "bind9" to the file `/srv/fai/config/package_config/WEB` on our FAI host *goldmaster*. This will ensure that the preceding manual step never needs to be performed when the host is reimaged.

We'll continue setting up *etchlamp* manually to ensure that we know the exact steps to configure an internal DNS server. Once we're done, we'll automate the process using cfengine. Note that the `bind9` package creates a user account named "bind." Add the lines from your `passwd`, `shadow`, and `group` files to your standardized Debian account files in cfengine. We'll also have to set up file-permission enforcement using cfengine. The BIND installation process might pick different user ID (UID) or group ID (GID) settings from the ones we'll copy out using cfengine.

The Debian `bind9` package stores its configuration in the `/etc/bind` directory. The package maintainer set things up in a flexible manner, where the installation already has the standard and required entries in `/etc/bind/named.conf`, and the configuration files use an `include` directive to read two additional files meant for site-specific settings:

- `/etc/bind/named.conf.options`: You use this file to configure the options section of `named.conf`. The options section is used to configure settings such as the name server's working directory, recursion settings, authentication-key options, and more. See the relevant section of the BIND 9 Administrator's Reference Manual for more information: `http://www.isc.org/sw/bind/arm95/Bv9ARM.ch06.html#options`.

- `/etc/bind/named.conf.local`: This file is meant to list the local zones that this BIND instance will load and serve to clients. These can be zone files on local disk, zones slaved from another DNS server, forward zones, or stub zones. We're simply going to load local zones, making this server the "master" for the zones in question.

The existence of these files means that we don't need to develop the configuration files for the standard zones needed on a BIND server; we need only to synchronize site-specific zones. Here is the `named.conf.options` file as distributed by Debian:

```
options {
        directory "/var/cache/bind";

        // If there is a firewall between you and nameservers you want
        // to talk to, you might need to uncomment the query-source
        // directive below.  Previous versions of BIND always asked
        // questions using port 53, but BIND 8.1 and later use an unprivileged
        // port by default.

        // query-source address * port 53;

        // If your ISP provided one or more IP addresses for stable
        // nameservers, you probably want to use them as forwarders.
        // Uncomment the following block, and insert the addresses replacing
        // the all-0's placeholder.

        // forwarders {
        //      0.0.0.0;
        // };

        auth-nxdomain no;    # conform to RFC1035
        listen-on-v6 { any; };
};
```

The only modification we'll make to this file is to change the `listen-on-v6` line to this:

```
listen-on-v6 { none; };
```

Because we don't intend to utilize IPv6, we won't have BIND utilize it either.

The default Debian `/etc/bind/named.conf.local` file has these contents:

```
//
// Do any local configuration here
//

// Consider adding the 1918 zones here, if they are not used in your
// organization
//include "/etc/bind/zones.rfc1918";
```

Note the `zones.rfc1918` file. It is a list of "private" IP address ranges specified in RFC1918. The file has these contents:

```
zone "10.in-addr.arpa"      { type master; file "/etc/bind/db.empty"; };

zone "16.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "17.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "18.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "19.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "20.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "21.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "22.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "23.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "24.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "25.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "26.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "27.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "28.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "29.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "30.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };
zone "31.172.in-addr.arpa"  { type master; file "/etc/bind/db.empty"; };

zone "168.192.in-addr.arpa" { type master; file "/etc/bind/db.empty"; };
```

It is a good idea to include this configuration file, with an important caveat we'll cover later. When you use this file, the db.empty zone file is loaded for all the RFC1918 address ranges. And because those are valid zone files with no entries for individual reverse DNS records (i.e., PTR records), the DNS traffic for those lookups won't go out to the public DNS. A "host not found" response will be returned to applications looking up the PTR records for IPs in those ranges. Those IP ranges are intended only for private use, so the DNS traffic for these networks should stay on private networks. Most sites utilize those ranges, so the public DNS doesn't have a set of delegated servers that serves meaningful information for these zones.

The caveat mentioned earlier is that we will not want to serve the db.empty file for the 192.168.x.x range that we use at our site. This means we'll delete this line from zones.rfc1918:

```
zone "168.192.in-addr.arpa" { type master; file "/etc/bind/db.empty"; };
```

Then we'll uncomment this line in /etc/bind/named.conf.local by deleting the two slashes at the start of the line:

```
//include "/etc/bind/zones.rfc1918";
```

Next, you'll need to create the *campin.net* and 168.192.in-addr.arpa zone files. The file /etc/bind/db.campin.net has these contents:

```
$TTL 600
@               IN      SOA     etchlamp.campin.net. hostmaster.campin.net. (
                                2008072900 ; serial
                                1800        ; refresh (30 minutes)
                                600         ; retry (10 minutes)
                                2419200     ; expire (4 weeks)
                                600         ; minimum (10 minutes)
                                )

                IN      NS      etchlamp.campin.net.

; the A record for campin.net
        600     IN      A       66.219.68.159

etchlamp        IN      A       192.168.1.239
aurora          IN      A       192.168.1.248
goldmaster      IN      A       192.168.1.249
rhmaster        IN      A       192.168.1.251
rhlamp          IN      A       192.168.1.236
hemingway       IN      A       192.168.1.237

; www.campin.net is a CNAME back to the A record for campin.net
www     600     IN      CNAME   @

skitzo  86400   IN      A       64.81.57.165
scampi  86400   IN      A       66.219.68.159

; www.campin.net is a CNAME back to the A record for campin.net
www     600     IN      CNAME   @

; give the default gateway an easy to remember name
gw              IN      A       192.168.1.1
```

We created entries for our six hosts, our local gateway address, and some records from our public zone.

Next, you need to create the "reverse" zone, in the file /etc/bind/db.192.168:

```
$TTL    600
@               IN      SOA     etchlamp.campin.net. hostmaster.campin.net. (
                                2008072900 ; serial
                                1800       ; refresh (30 minutes)
                                600        ; retry (10 minutes)
                                2419200    ; expire (4 weeks)
                                600        ; minimum (10 minutes)
                                )

@       IN      NS      etchlamp.campin.net.

$ORIGIN 1.168.192.in-addr.arpa.

1       IN      PTR     gw.campin.net.

236     IN      PTR     rhlamp.campin.net.
237     IN      PTR     hemingway.campin.net.
239     IN      PTR     etchlamp.campin.net.
248     IN      PTR     aurora.campin.net.
249     IN      PTR     goldmaster.campin.net.
251     IN      PTR     rhmaster.campin.net.
```

The `$ORIGIN` keyword set all the following records to the `192.168.1.0/24` subnet's `in-addr.arpa` reverse DNS range. This made the records simpler to type in. Be sure to terminate the names on the right-hand side of all your records with a dot (period character) when you specify the fully qualified domain name.

Next, populate the file `/etc/bind/named.conf.local` with these contents, to utilize our new zone files:

```
include "/etc/bind/zones.rfc1918";

zone "campin.net" {
        type master;
        file "/etc/bind/db.campin.net";
};

zone "168.192.in-addr.arpa" {
        type master;
        file "/etc/bind/db.192.168";
};
```

Restart BIND using the included init script:

```
# /etc/init.d/bind9 restart
```

Look for errors from the init script, as well as in the /var/log/daemon.log log file. If the init script successfully loaded the zones, you'll see lines like this in the log file:

```
Jul 29 17:43:30 etchlamp named[2580]: zone 168.192.in-addr.arpa/IN: loaded serial
2008072900
Jul 29 17:43:30 etchlamp named[2580]: zone campin.net/IN: loaded serial 2008072900
Jul 29 17:43:30 etchlamp named[2580]: running
```

Test resolution from another host on the local subnet using the dig command:

```
$ dig @etchlamp gw.campin.net.

; <<>> DiG 9.3.4-P1.1 <<>> @etchlamp gw.campin.net.
; (1 server found)
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45274
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;gw.campin.net.                 IN      A

;; ANSWER SECTION:
gw.campin.net.         600     IN      A       192.168.1.1

;; AUTHORITY SECTION:
campin.net.            600     IN      NS      etchlamp.campin.net.

;; ADDITIONAL SECTION:
etchlamp.campin.net.   600     IN      A       192.168.1.239

;; Query time: 19 msec
;; SERVER: 192.168.1.239#53(192.168.1.239)
;; WHEN: Tue Jul 29 17:45:49 2008
;; MSG SIZE  rcvd: 86
```

This query returns the correct results. In addition, the flags section of the response has the aa bit set, meaning that the remote server considers itself authoritative for the records it returns. Do the same thing again, but this time query for a reverse record:

```
$ dig @etchlamp -x 192.168.1.1 ptr

; <<>> DiG 9.3.4-P1.1 <<>> @etchlamp -x 192.168.1.1 ptr
; (1 server found)
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47489
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;1.1.168.192.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
1.1.168.192.in-addr.arpa. 600   IN      PTR     gw.campin.net.

;; AUTHORITY SECTION:
168.192.in-addr.arpa.   600     IN      NS      etchlamp.campin.net.

;; ADDITIONAL SECTION:
etchlamp.campin.net.    600     IN      A       192.168.1.239

;; Query time: 2 msec
;; SERVER: 192.168.1.239#53(192.168.1.239)
;; WHEN: Tue Jul 29 17:46:11 2008
;; MSG SIZE  rcvd: 108
```

Again, we have successful results. We had to modify only three included files (`zones.rfc1918`, `named.conf.local`, and `named.conf.options`), and create two new ones (`db.campin.net` and `db.192.168`). Now we know the file locations and file contents that we need in order to host our private DNS on a Debian system running BIND.

Automating the BIND Configuration

We'll create a cfengine task to distribute our BIND configuration, and as usual it will restart the BIND daemon when the configuration files are updated.

Here are the steps to automate this process:

1. Copy the BIND configuration files and zone files (that we created during the development process on *etchlamp*) to the cfengine master.

2. Create a cfengine task that copies the BIND configuration files and zones, and restarts the BIND daemon when the files are copied.

3. Define a new "DNS server" role in cfengine using a class.

4. Create a new hostgroup file for this new server role in cfengine.

5. Import the new task into the new DNS server hostgroup file in cfengine.

6. Import the new hostgroup file into `cfagent.conf`, so that the hostgroup and task are used.

7. Test out the entire automation process for the DNS server role by reimaging the DNS server host.

The first step is to get our files from *etchlamp* onto the cfengine master, in the correct location. Create the directory on *goldmaster*:

```
# mkdir -p /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/bind/debian-ext
```

Now copy those five files from *etchlamp* to the new directory on *goldmaster*:

```
# pwd
/etc/bind
# scp zones.rfc1918 named.conf.local db.campin.net db.192.168 named.conf.options \
 goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/bind/debian-ext/
```

Name the task `PROD/inputs/tasks/apps/bind/cf.debian_external_cache` and start the task with these contents:

```
groups:
        have_etc_rndc_key       = ( FileExists(/etc/bind/rndc.key) )
```

Later in this task we'll perform permission fixes on the `rndc.key` file, but we like to make sure it's actually there before we do it.

We'll continue explaining the `cf.debian_external_cache` task. In the `control` section we tell cfengine about some classes that we dynamically define, and put in an entry for `DefaultPkgMgr`:

```
control:
        any::
                addinstallable          = (      bind_installed bind_installed
                                                            reload_bind
                                        )

        debian::
                DefaultPkgMgr           = ( dpkg )
```

which is required when we use the `packages` action:

```
packages:
    debian::
        bind9
            version=9.3.4
            cmp=ge
            define=bind_installed
            elsedefine=bind_not_installed
```

We use the `packages` action simply to detect whether the `bind9` package is installed, and we go with the version installed by Debian 4.0 ("Etch") as the minimum installed version. Assumptions will only lead to errors, so we double-check even basic assumptions such as whether BIND has been installed on the system at all.

Here we use the `processes` action to start up BIND when it is missing from the process list, but only if it's one of our external caches, and only if the `bind9` package is installed:

```
processes:
    debian.bind_installed::
        "named" restart "/etc/init.d/bind9 start" inform=false umask=022
```

There's no point in even trying to start BIND if it isn't installed.

Here we copy the five files we placed into the `debian-ext` directory to the host's `/etc/bind` directory:

```
copy:
    debian.bind_installed::
        $(master_etc)/bind/debian-ext/
            dest=/etc/bind/
            r=inf
            mode=644
            type=checksum
            purge=false
            server=$(fileserver)
            encrypt=true
            owner=root
            group=root
            define=reload_bind
```

We carefully named the source directory `debian-ext` because we might end up deploying BIND to our Debian hosts later in some other configuration. Having a complete source directory to copy makes the `copy` stanza simpler. We know that only the files

we want to overwrite are in the source directory on the cfengine master—so be careful not to add files into the source that you don't want automatically copied out. You also have to be careful not to purge during your copy, or you'll lose all the default Debian bind9 configuration files you depend on.

This shellcommands section uses the reload_bind class to trigger a restart of the BIND daemon:

```
shellcommands:
        debian.restart_bind::
                # when the config is updated, reload bind
                "/etc/init.d/bind9 reload" timeout=30
```

The reload_bind class is defined when files are copied from the master, via the define= line.

These file and directory settings fix the important BIND files and directory permissions in the unlikely event that the bind user's UID and GID change:

```
files:
        debian.bind_installed.have_etc_rndc_key::
                /etc/bind/rndc.key owner=bind group=bind m=640 action=fixall
                                              inform=true syslog=on


directories:
        debian.bind_installed::
                /var/cache/bind mode=775 owner=root group=bind inform=true syslog=on
                /etc/bind mode=2755 owner=root group=bind inform=true syslog=on
```

Such an event happens if and when we later synchronize all the user accounts across our site. Now we'll take steps to recover properly from a bind-user UID/GID change. Set up an alerts section to issue a warning when you designate a host as an external_debian_bind_cache but don't actually have the bind9 package installed:

```
alerts:
        debian.!bind_installed::
                "Error: I am an external cache but I don't have bind9 installed."
```

We use the packages action in this task, so we need to add packages to the actionsequence in the control/cf.control_cfagent_conf file for cfengine to run it:

```
actionsequence = (
    directories
    disable
      packages
    copy
    editfiles
    links
    files
    processes
    shellcommands
)
```

Now we need to add the task to a hostgroup file, but it certainly isn't a good fit for
the `cf.any` hostgroup. Create a new hostgroup file for the task and place it at `PROD/inputs/
hostgroups/cf.external_dns_cache`. That name was chosen carefully; we won't assume that
all our caching DNS servers will be running Debian, or even BIND for that matter. The
role is to serve DNS to our network, and the hostgroup name is clear about that. The con-
tents of this new hostgroup file are:

```
import:
      any::
                tasks/app/bind/cf.debian_external_cache
```

Now we need to define an alias for the hosts that serve this role. We'll edit `PROD/
inputs/classes/cf.main_classes` and add this line:

```
caching_dns_servers     = (    etchlamp )
```

Then we'll edit `cfagent.conf` and add an import for the new hostgroup file for the
`caching_dns_servers` class:

```
caching_dns_servers::             hostgroups/cf.external_dns_cache
```

Wait! If you were to run `cfagent -qv` on *etchlamp* at this point, the file `PROD/inputs/
hostgroups/cf.external_dns_cache` would not be imported, even though `cfagent`'s
"Defined Classes" output shows that the `caching_dns_servers` class is set. Most people
learn this important lesson the hard way, and we wanted you to learn it the hard way as
well, so it will be more likely to stick.

IMPORTS IN CFENGINE

If a cfengine configuration file uses imports, then the entire file needs to be made up of imports. You cannot use classes in the *importing* file that are defined in the *imported* file.

We encountered the second point when we imported the file `classes/cf.main_classes` from `cfagent.conf`, then tried to use the class `caching_dns_servers` in `cfagent.conf`. This doesn't work, because cfengine reads in the imported files only after the main file is completely parsed. We'll need to do our hostgroup mappings in an imported file as well, and we'll reorganize our `PROD/inputs` directory just a little bit to compensate.

To reorganize in a way that will work with cfengine's issues around imports but preserve our hostgroup system, delete these two lines from `cfagent.conf`:

```
any::                   hostgroups/cf.any
caching_dns_servers::   hostgroups/cf.external_dns_cache
```

Place the line in a new file, `hostgroups/cf.hostgroup_mappings`, with these contents:

```
import:
    any::                           hostgroups/cf.any
    caching_dns_servers::           hostgroups/cf.external_dns_cache
```

Remember that any lines added below the `cf.external_dns_cache` import will apply only to the `caching_dns_servers` class, unless a new class is specified. That is a common error made by inexperienced cfengine-configuration authors, and often even experienced ones.

We need to add the `cf.hostgroup_mappings` file to `cfagent.conf`, by adding this line at the end:

```
hostgroups/cf.hostgroup_mappings
```

We don't need to specify the `any::` class because it's already inherent in all of this task's imports. In fact, unless otherwise specified, it's inherent in every cfengine action.

Now we should validate that our hostgroup is being imported properly—by running `cfagent -qv` on *etchlamp*. Look for this line in the output:

```
Looking for an input file tasks/app/bind/cf.debian_external_cache
```

Success! All future hostgroup imports will happen from the `cf.hostgroup_mappings` file. We'll mention one last thing while on the subject of imports. Note that we don't do any imports in any of our task files. Any file containing actions other than `import` should

not use the `import` action at all. You can get away with this if you do it carefully, but we'll avoid it like the plague.

Remember that every host that ever matches the `caching_dns_servers` class will import the `cf.external_dns_cache hostgroup` file, and therefore will also import the `cf.debian_external_cache` task. If a Solaris host is specified as a member of the `caching_dns_servers` class, it will not do anything unintended when it reads the `cf.debian_external_cache` task. This is because we specify the `debian` class for safety in the class settings for all our actions. You could further protect non-Debian hosts by importing the task only for Debian hosts from the `hostgroups/cf.external_dns_cache` file:

```
import:
        debian::
                tasks/app/bind/cf.debian_external_cache
```

Importing the task this way is safer, but even if you do, you should make sure that your cfengine configuration files perform actions only on the hosts you intend. Always be defensive with your configurations, and you'll avoid unintended changes. Up until this point, we have purposely made our task files safe to run on any operating system and hardware architecture by limiting the cases when an action will actually trigger, and we will continue to do so.

Now it's time to reimage *etchlamp* via FAI, and make sure that the DNS service is fully configured and working when we set up *etchlamp* from scratch. Always ensure that your automation system works from start to finish. The *etchlamp* host's minimal install and configuration work will take under an hour, so the effort and time is well worth it.

While *etchlamp* is reimaging, remove the old installation's cfengine public key on the cfengine master because the reimaging process will generate a new key. The host *etchlamp* has the IP `192.168.1.239`, so run this command on *goldmaster* as the `root` user:

```
# rm /var/lib/cfengine2/ppkeys/root-192.168.1.239.pub
```

When *etchlamp* reboots after installation, the cfengine daemons don't start up because we have only the bootstrap `update.conf` and `cfagent.conf` files in `/var/lib/cfengine2/inputs`. We need to make sure that `cfagent` runs once upon every reboot. Modify `/srv/fai/config/scripts/FAIBASE/50-cfengine` on the FAI server to add a line that will run `cfagent` upon every boot, mainly to help on the first boot after installation:

```
#! /usr/sbin/cfagent -f

control:
    any::
    actionsequence = ( editfiles )
    EditFileSize = ( 30000 )
```

```
editfiles:
    any::
        { ${target}/etc/aliases
          AutoCreate
          AppendIfNoSuchLine "root: nate@campin.net"
        }

        { ${target}/etc/default/cfengine2
          ReplaceAll "=0$" With "=1"
        }

        { ${target}/etc/init.d/bootmisc.sh
          AppendIfNoSuchLine "/usr/sbin/cfagent -qv"
        }
```

This configures the cfagent program to run from the /etc/init.d/bootmisc.sh file at boot time. So, to recap: We started another reimage of *etchlamp* and removed /var/lib/cfengine2/ppkeys/root-192.168.1.239.pub again on the cfengine master while the host was reimaging.

The host *etchlamp* returned from reimaging fully configured, with cfengine running. Now every time a Debian host boots at our site after FAI installs it, it will run cfagent during boot. Without logging into the host (i.e., without manual intervention), you can run a DNS query against *etchlamp* successfully:

```
$ dig @etchlamp gw.campin.net

; <<>> DiG 9.3.4-P1.1 <<>> @etchlamp gw.campin.net
; (1 server found)
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 59779
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;gw.campin.net.                  IN      A

;; ANSWER SECTION:
gw.campin.net.          600     IN      A       192.168.1.1

;; AUTHORITY SECTION:
campin.net.             600     IN      NS      etchlamp.campin.net.
```

```
;; ADDITIONAL SECTION:
etchlamp.campin.net.    600    IN    A       192.168.1.239

;; Query time: 1 msec
;; SERVER: 192.168.1.239#53(192.168.1.239)
;; WHEN: Wed Jul 30 00:39:52 2008
;; MSG SIZE  rcvd: 86
```

What we have accomplished here is worth celebrating. If you suffer total system failure on the host *etchlamp*, you can simply reimage a new host with the same hostname and bring it back onto the network as a DNS server. This is exactly what we want of all hosts at our site. As you deploy web servers, NFS servers, and other system roles, you should test that the host can be reimaged and properly configured to serve its designated function again without any human intervention. The extent of human involvement should be to identify hardware and do any Kickstart/FAI/JumpStart configuration needed to support imaging that piece of hardware.

We have a private DNS server now, and although it's the only one, we'll configure the /etc/resolv.conf files across all our hosts to utilize the new DNS server before any other DNS servers. We'll still list our existing DNS server, 192.168.1.1, as the second nameserver in /etc/resolv.conf in case *etchlamp* becomes unreachable.

Cfengine has a resolve action that you can use to configure the /etc/resolv.conf file. We'll create a task called tasks/os/cf.resolv_conf and test whether we have resolv.conf in a directory where postfix is chrooted by default on Debian:

```
classes:
        have_postfix_resolv  = ( FileExists(/var/spool/postfix/etc/resolv.conf) )
```

Here's something we've never done before—change the actionsequence in a task file:

```
control:
        any::
                addinstallable           = ( reloadpostfix )
                actionsequence         = ( resolve )
                EmptyResolvConf     = (true )
```

The preceding code adds resolve to the actionsequence. We can add it to the global actionsequence defined in the control/cf.control_cfagent_conf file that's imported directly from cfagent.conf, but there's really no need. We'll generally add actionsequence items there, but we wanted to demonstrate that we still have some flexibility in our cfengine configurations.

The order of the IP addresses and comment is preserved in the `/etc/resolv.conf` file:

```
resolve:
      any::
                  #
                  # If EmptyResolvConf is set to true, we'll completely wipe out
                  # resolv.conf EVEN if we have no matches in the below classses!
                  #
                  # When EmptyResolvConf is set, always be sure that you have an
                  # any class to catch all hosts with some basic nameserver entries.
                  #
                  192.168.1.239
                  192.168.1.1
                  "# resolv.conf edited by cfengine, don't muck with this"
```

We added the comment so that if any SAs want to change `/etc/resolv.conf` directly with a text editor, they'll realize that the file is under cfengine control.

We use the local copy to keep postfix name resolution working properly after cfengine updates the `/etc/resolv.conf` file and to restart postfix when we do the copy:

```
copy:

      # this is a local copy to keep the chroot'd postfix resolv.conf up to date
      have_postfix_resolv::
                  /etc/resolv.conf
                          dest=/var/spool/postfix/etc/resolv.conf
                          mode=644
                          owner=root
                          group=root
                          type=checksum
                          define=reloadpostfix

shellcommands:
      # reload postfix when we update the chroot resolv.conf
      debian.reloadpostfix::
                  "/etc/init.d/postfix restart" timeout=30 inform=true
```

Next, add the task to `PROD/inputs/hostgroups/cf.any`. Once the task is enabled, we connect to the host *aurora* and inspect the new /etc/resolv.conf:

```
# cat /etc/resolv.conf
domain campin.net
nameserver 192.168.1.239
nameserver 192.168.1.1
# resolv.conf edited by cfengine, don't muck with this
```

Then test name resolution:

```
# nslookup gw
Server:        192.168.1.239
Address:       192.168.1.239#53

Name:   gw.campin.net
Address: 192.168.1.1
```

We're done with the DNS for now. When we get more hardware to deploy another Debian-based DNS server system, we'll add it to the `caching_dns_servers` class, let cfengine set up BIND, then update `cf.resolv_conf` to add another `nameserver` entry to all our site's `/etc/resolv.conf` files.

Taking Control of User Account Files

We need to take control of the user accounts at our site. Every site eventually needs a centralized mechanism the SA staff can use to create and delete accounts, lock them out after a designated number of failed logins, and log user access. This will be usually a system such as NIS/NIS+, LDAP, or perhaps LDAP combined with Kerberos.

At this point, we're not talking about setting up a network-based authentication system—we're not ready for that yet. First, we need to take control of our local account files: `/etc/passwd`, `/etc/shadow`, and `/etc/group`. Even if we already had LDAP deployed at our site and all our users had accounts only in the LDAP directory, we would need to be able to change the local `root` account password across all our systems on a regular basis. In addition, we normally change the default shell on many system accounts that come with the system, for added security. Allowing local account files to go unmanaged is a security risk.

Standardizing the Local Account Files

We have three different sets of local account files at our site: those for Red Hat, Solaris, and Debian. We're going to standardize the files for each system type, and synchronize those files to each system from our central cfengine server on a regular basis. Over time,

we'll need to add accounts to the standard files to support new software (e.g., a "mysql" user to run the MySQL database software). We will never add them directly onto the client systems; instead, we will add them to the centralized files.

We have only two installed instances of each OS type, so it's easy to copy all the files to a safe location and consolidate them. Because we're copying the shadow files, the location should be a directory with restrictive permissions:

```
# mkdir -m 700 /root/authfiles
# cd /root/authfiles
# for host in goldmaster rhmaster rhlamp ethlamp hemingway aurora ; \
do for file in passwd shadow group ; do [ -d $file ] || mkdir -m 700 $file ; \
scp root@${host}:/etc/$file ${file}/${file}.$host ; done ; done
```

These commands will iterate over all our hosts and copy the three files we need to a per-file subdirectory, with a file name that includes the hostname of the system that the file is from. We will illustrate standardization of account files for our two Solaris hosts only, to keep this section brief. Assume that we will perform the same process for Debian and Red Hat.

Now you can go into each directory and compare the files from the two Solaris hosts:

```
# cd /root/authfiles/passwd
# diff passwd.aurora passwd.hemingway
12a13,14
> postgres:x:90:90:PostgreSQL Reserved UID:/:/usr/bin/pfksh
> svctag:x:95:12:Service Tag UID:/:
```

The *hemingway* host has two accounts that weren't created on *aurora*. We won't need the postgres account, used to run the freeware Postgres database package. We will keep the svctag account because the Solaris serial port–monitoring facilities use it.

```
# mv passwd.hemingway passwd.solaris
# rm passwd.aurora
```

Edit passwd.solaris and remove the line starting with postgres. Now the passwd.solaris file contains the accounts we need on both systems. We will use this as our master Solaris password file.

Go through the same procedure for the Solaris shadow files:

```
# cd ../shadow
# diff shadow.hemingway shadow.aurora
13,14d12
< postgres:NP:::::::
< svctag:*LK*:6445::::::
# mv shadow.hemingway shadow.solaris
# rm shadow.aurora
```

Use a text editor to remove the `postgres` line from `shadow.solaris` as well.
Here's the procedure for the `group` file:

```
# diff group.hemingway group.aurora
17d16
< postgres::90:
20a20
> sasl::100:
```

We have a `postgres` group on *hemingway* that we'll remove, and a `sasl` group on
aurora that we'll keep. SASL is the Simple Authentication and Security Layer, which you
use to insert authentication into network protocols. We might end up needing this if we
set up authenticated Simple Mail Transfer Protocol (SMTP) or another authenticated net-
work protocol later on.

```
# mv group.aurora group.solaris
# rm group.hemingway
```

Now we'll move our new files into the directories we created for these files (back
when we originally created our `masterfiles` directory in Chapter 5).

```
# scp group/group.solaris \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/group/
# scp passwd/passwd.solaris \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/passwd/
# scp shadow/shadow.solaris \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/shadow/
```

Now perform the same decision-making process for the Red Hat and Debian account
files. When you're done, move them into the proper place in the `masterfiles` directories
as you did for the Solaris account files. You need to be careful during this stage that you
don't change the UID or GID of system processes without setting up some remediation
steps in cfengine.

Our two Debian systems ended up with different UID and GID numbers for the post-
fix user and group, as well as for the `postdrop` group (also used by postfix). We chose to

stick with the UID and GID from the *goldmaster* host, and to add some permission fixes in a cfengine task that will fix the ownership of the installed postfix files and directories.

THE VARYING UID AND GID NUMBERS IN DEBIAN LINUX

We really wish Debian had designated UID/GID numbers for all system user accounts, and therefore consistent UIDs across all installations. In our opinion, we shouldn't have to fix this problem—the OS vendor should deal with it for us. To be fair, the Debian Policy Manual specifies some standard and globally identical UID/GID settings for a limited set of base system accounts, and a dynamic range for other system accounts such as the BIND and NTP users. This surely means less maintenance for a project as large as Debian, but it means a fair amount of pain for us to deal with it.

Once we've standardized all our files, we have these files on the cfengine master system:

```
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/etc
# ls passwd/ shadow/ group
group:
./  ../  group.debian  group.redhat  group.solaris

passwd/:
./  ../  passwd.debian  passwd.redhat  passwd.solaris

shadow/:
./  ../  shadow.debian  shadow.redhat  shadow.solaris
```

Distributing the Files with cfengine

We'll develop a cfengine task to distribute our new master account files. We will add some safety checks into this task because we need to treat these files with the utmost caution.

We'll place the file in a task called `cf.account_sync`, with these contents:

```
classes: # synonym groups:
            safe_to_sync    = (      debian_4_0
                                        redhat_s_5_2
                                        sunos_5_10
                                    )
```

We create a group to control which classes of systems get the account-file synchronization. These three classes encompass all the systems we're currently running at our site. We do this because we know our account files will work on the UNIX/Linux versions that we're currently running, but we don't know if they will work on older or newer versions. In fact, if you don't know for sure that something will work, you should assume that it won't.

So if you deploy a new type of system at your site, you run the risk that the new system type won't have local account files synchronized by cfengine. Take measures to detect this situation in the task, and alert the site administrators:

```
control:
        debian::
                passwd_file   = ( "passwd.debian" )
                shadow_file   = ( "shadow.debian" )
                group_file    = ( "group.debian" )


        redhat::
                passwd_file   = ( "passwd.redhat" )
                shadow_file   = ( "shadow.redhat" )
                group_file    = ( "group.redhat" )


        solaris|solarisx86::
                passwd_file   = ( "passwd.solaris" )
                shadow_file   = ( "shadow.solaris" )
                group_file    = ( "group.solaris" )
```

Here you'll recognize the standardized files we created earlier.

```
copy:
        safe_to_sync::
                $(master_etc)/passwd/$(passwd_file)
                        dest=/etc/passwd
                        mode=644
                        server=$(fileserver)
                        trustkey=true
                        type=checksum
                        owner=root
                        group=root
                        encrypt=true
                        verify=true
                        size=>512
```

```
$(master_etc)/shadow/$(shadow_file)
        dest=/etc/shadow
        mode=400
        owner=root
        group=root
        server=$(fileserver)
        trustkey=true
        type=checksum
        encrypt=true
        size=>200

$(master_etc)/group/$(group_file)
        dest=/etc/group
        mode=644
        owner=root
        group=root
        server=$(fileserver)
        trustkey=true
        type=checksum
        encrypt=true
        size=>200
```

The size keyword in these copy stanzas adds file-size minimums for the passwd, shadow, and group file copies. We use this keyword so we don't copy out empty or errone-ously stripped down files. The minimums should be around half the size of the smallest version that we have of that particular file. You might need to adjust the minimums if the files happen to shrink later on. Usually these files grow in size.

Here we define an alert for hosts that don't have local account files to synchronize:

```
alerts:
        !safe_to_sync::
                "I am not set up to sync my account files, please check on it."
```

The alerts action simply prints text used to alert the system administrator. The cfexecd daemon will e-mail this output.

Next, put the task into the cf.any hostgroup:

```
import:
        any::
                tasks/os/cf.motd
                tasks/os/cf.cfengine_cron_entries
                tasks/os/cf.ntp
                tasks/os/cf.account_sync
```

When `cfagent` performs a copy, and the `repository` variable is defined, the version of the file before the copy is backed up to the `repository` directory. Define `repository` like this in `PROD/inputs/control/cf.control_cfagent_conf`:

```
repository              = ( $(workdir)/backups )
```

This means you can see the old local account files in the backup directory on each client after the copy. On Debian the directory is `/var/lib/cfengine2/backups`, and on the rest of our hosts it's `/var/cfengine/backups`.

If you encounter any problems, compare the previous and new versions of the files, and see if you left out any needed accounts. Be aware that each performed copy overwrites previous backup files in the `repository` directory. This means you'll want to validate soon after the initial sync. We also saved the original files in the home directory for the `root` user. It's a good idea to store them for at least a few days in case you need to inspect them again.

Our *etchlamp* system had the postfix account's UID and GID change with this local account sync. The GID of the `postdrop` group also changed. We can fix that with cfengine, in a task we call `cf.postfix_permissions`:

```
classes:  # synonym groups:
        have_var_spool_postfix         = ( IsDir("/var/spool/postfix") )
        have_var_spool_postfix_public  = ( IsDir("/var/spool/postfix/public") )
        have_var_spool_postfix_maildrop = ( IsDir("/var/spool/postfix/maildrop") )
        have_usr_sbin_postdrop         = ( IsDir("/usr/sbin/postdrop") )
        have_usr_sbin_postqueue        = ( IsDir("/usr/sbin/postqueue") )
```

Here we have some classes based on whether files or directories are present on the system. We don't want to assume that postfix is installed on the system. We previously added postfix into the list of FAI base packages, but we can't guarantee with absolute certainty that every Debian system we ever manage will be running postfix.

We could use a more sophisticated test, such as verifying that the postfix Debian package is installed, but a simple directory test suffices and happens quickly:

```
directories:
        debian.have_var_spool_postfix_public::
                /var/spool/postfix/public mode=2710
                            owner=postfix group=postdrop inform=true

        debian.have_var_spool_postfix_maildrop::
                /var/spool/postfix/maildrop mode=1730
                            owner=postfix group=postdrop inform=true
```

```
debian.have_var_spool_postfix::
        /var/spool/postfix/active mode=700 owner=postfix
                                              group=root inform=true
        /var/spool/postfix/bounce mode=700 owner=postfix
                                                group=root inform=true
        /var/spool/postfix/corrupt mode=700 owner=postfix
                                                group=root inform=true
        /var/spool/postfix/defer mode=700 owner=postfix
                                              group=root inform=true
        /var/spool/postfix/deferred mode=700 owner=postfix
                                                  group=root inform=true
        /var/spool/postfix/flush mode=700 owner=postfix
                                              group=root inform=true
        /var/spool/postfix/hold mode=700 owner=postfix
                                            group=root inform=true
        /var/spool/postfix/incoming mode=700 owner=postfix
                                                  group=root inform=true
        /var/spool/postfix/private mode=700 owner=postfix
                                              group=root inform=true
        /var/spool/postfix/trace mode=700 owner=postfix
                                            group=root inform=true
```

Here we make sure that all the postfix spool directories have the correct ownership and permissions. If you blindly create the directories without verifying that /var/spool/ postfix is already there, it'll appear as if postfix is installed when it isn't. This might seem like a minor detail, but the life of an SA comprises a large collection of minor details such as this. Creating confusing situations such as unused postfix spool directories is just plain sloppy, and you should avoid doing so.

Here we ensure that two important postfix binaries have the SetGID bit set, as well as proper ownership:

```
files:
      debian.have_usr_sbin_postqueue::
            /usr/sbin/postqueue m=2555 owner=root  group=postdrop
                                            action=fixall inform=true


      debian.have_usr_sbin_postdrop::
            /usr/sbin/postdrop m=2555 owner=root  group=postdrop
                                          action=fixall inform=true
```

At any time you can validate that postfix has the proper permissions by executing this line:

```
# postfix check
```

You'll also want to restart any daemons that had their process-owner UID change after you fixed file and directory permissions.

Now we'll put the task into the `cf.any` hostgroup:

```
import:
        any::
                tasks/os/cf.motd
                tasks/os/cf.cfengine_cron_entries
                tasks/os/cf.ntp
                tasks/os/cf.account_sync
                tasks/os/cf.postfix_permissions
```

You're probably wondering why we put the `cf.postfix_permissions` task into the `cf.any` hostgroup, when it performs actions only on Debian hosts. We did this because we might end up having to set postfix permissions on other platforms later. The task does nothing on host types for which it's not intended, so you face little risk of damage.

From this point on, when you install new packages at your site that require additional local system accounts, manually install on one host (of each platform) as a test. When you (or the package) find the next available UID and GID for the account, you can add the account settings into your master `passwd`, `shadow`, and `group` files for synchronization to the rest of your hosts. That way, when you deploy the package to all hosts via cfengine, the needed account will be in place with the proper UID and GID settings. This is another example of how the first step in automating a procedure is to make manual changes on test systems.

Adding New User Accounts

Now you can add user accounts at your site. We didn't want to add a single user account before we had a mechanism to standardize UIDs across the site. The last thing we need is to deploy LDAP or a similar service later on, and have a different UID for each user account—on many systems. We have avoided that mess entirely.

At this point, you can simply add users into the centralized account files stored on the cfengine master. New users won't automatically have a home directory created, but later in the chapter we'll address that issue using a custom `adduser` script, an NFS-mounted home directory, and the automounter.

Using Scripts to Create User Accounts

You shouldn't ever create user accounts manually by hand-editing the centralized `passwd`, `shadow`, and `group` files at your site. We'll create a simple shell script that chooses the next available UID and GID, prompts for a password, and properly appends the account information into the account files.

We'll make the script simple because we don't intend to use it for long. Before we even write it, we need to consider where we'll put it. We know that it is the first of what will surely be many administrative scripts at our site. When we first created the `masterfiles` directory structure, we created the directory `PROD/repl/admin-scripts/`, which we'll put into use now.

We'll copy the contents of this directory to all hosts at our site, at a standard location. We've created a cfengine task to do this, called `cf.sync_admin_scripts`:

```
copy:
        any::
                $(master)/repl/admin-scripts
                        dest=/opt/admin-scripts
                        mode=550
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        r=inf
                        purge=true


directories:
        any::
                /opt/admin-scripts  mode=750 owner=root group=root inform=false
```

We're copying every file in that directory, making sure each is protected from non-`root` users and executable only for members of the `root` group. Because we haven't set up special group memberships yet, SA staff will need to become `root` to execute these scripts—for now, anyway. Remember that our `actionsequence` specifies that `directories` runs before `copy`, so the directory will be properly created before the copy is attempted.

Add this entry to the end of the `cf.any` hostgroup:

```
tasks/misc/cf.sync_admin_scripts
```

You place the task in the `misc` directory because it's not application-specific and it doesn't affect part of the core operating system. Now you can utilize a collection of administrative scripts that is accessible across the site. You can create the new user script and place it in there. The script itself will have checks to make sure it is running on the appropriate master host.

We call the script `add_local_user`, and we don't append a file suffix such as `.sh`. This way, we can rewrite it later in Perl or Python and not worry about a misleading file suffix. UNIX doesn't care about file extensions, and neither should you.

```
#!/bin/sh
#############################################################################
# This script was written to work on Debian Linux, specifically the Debian host
# serving as the cfengine master at our site. Analysis should be done before
# attempting to run elsewhere.
#############################################################################
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/opt/admin-scripts

# this is the deepest shared directory for all the
#  passwd/shadow/group files
BASE_PATH=/var/lib/cfengine2/masterfiles/PROD/repl/root/etc
USERNAME_FILE=/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/USERFILE

case `hostname` in
goldmaster*)
        echo "This is the proper host on which to add users, continuing..."
        ;;
*)
        echo "This is NOT the proper host on which to add users, exiting now..."
        exit 1
        ;;
esac
```

We have only one cfengine master host that has the centralized files, so make sure we're running on the correct host before moving on. We also define a file, which we'll use later, to store usernames for accounts that we create:

```
cd $BASE_PATH

LOCKFILE=/root/add_user_lock

rm_lock_file() {
        rm -f $LOCKFILE
}

# don't ever run two of these at once
lockfile $LOCKFILE || exit 1
```

We define a file to use for locking to ensure that we run only one instance of this script at a time. We use methods that should prevent files from getting corrupted, but if two script instances copy an account file at the same time, update it, then copy it back into place, one of those instances will have its update overwritten.

Now collect some important information about the user account:

```
# We REALLY need to sanity check what we accept here, before blindly
# trusting the values, that's an excercise for the reader.
echo "Please specify a username for your new account, 8 chars or less: "
read USERNAME

echo "Please give the person's full name for your new account: "
read GECOS

stty -echo
echo "Please specify a password for your new account: "
read PASSWORD
stty echo
```

Later we should add some logic to test that the password meets certain criteria. The eight-character UNIX username limit hasn't applied for years on any systems that we run, but we observe the old limits just to be safe.

Here we generate an encrypted password hash for our shadow files:

```
ENC_PASS=`echo $PASSWD | mkpasswd -s`
```

You can add -H md5 to generate an MD5 hash, which is more secure. We've chosen to use the lowest common denominator here, in case we inherit some old system. Which type of hash you choose is up to you.

Now create the file containing the next available UID, if it doesn't already exist:

```
[ -f "$BASE_PATH/NEXTUID" ] || echo 1001 > $BASE_PATH/NEXTUID
```

Collect the UID and GID to use for the account. Always use the same number for both:

```
NEXTUID=`cat $BASE_PATH/NEXTUID`
```

Test that the value inside the NEXTUID file is numerically valid. We would hate to create an account with an invalid UID:

```
if [ -n "$NEXTUID" -a $NEXTUID -gt 1000 ]
then
        echo "Our next UID appears valid, continuing..."
else
        echo "The $BASE_PATH/NEXTUID file appears to be corrupt, please ➥
investigate."
        echo "Exiting now..."
        exit 1
fi
```

Here we set up the formatting of our account-file entries, to be used in the next section:

```
SEC_SINCE_EPOCH=`date +%s`
GROUP_FORMAT="$USERNAME:x:$NEXTUID:"
PASSWD_FORMAT="$USERNAME:x:$NEXTUID:$NEXTUID:$GECOS:/home/$USERNAME:/bin/bash"
SHADOW_FORMAT="$USERNAME:$ENC_PASS:$SEC_SINCE_EPOCH:7:180:14:7::"
```

If you use this script, you need to set values for the shadow fields that make sense at your site. The meanings are:

```
     1   login name
     2   encrypted password
     3   days since Jan 1, 1970 that password was last changed
     4   days before password may be changed
     5   days after which password must be changed
     6   days before password is to expire that user is warned
     7   days after password expires that account is disabled
     8   days since Jan 1, 1970 that account is disabled
     9   a reserved field (unused)
```

The script continues:

```
for groupfile in group/group*
do
      cp $groupfile ${groupfile}.tmp && \
      echo $GROUP_FORMAT >> ${groupfile}.tmp && \
      mv ${groupfile}.tmp $groupfile || \
      ( echo "Failed to update $groupfile - exiting now." ; rm_lock_file ; exit 1 )
done

for shadowfile in shadow/shadow*
do
      cp $shadowfile ${shadowfile}.tmp && \
      echo $SHADOW_FORMAT >> ${shadowfile}.tmp && \
      mv ${shadowfile}.tmp $shadowfile || \
      ( echo "Failed to update $shadowfile - exiting now." ; rm_lock_file ; ➡
exit 1 )

done

for passwdfile in passwd/passwd*
do
      cp $passwdfile ${passwdfile}.tmp && \
      echo $PASSWD_FORMAT >> ${passwdfile}.tmp && \
      mv ${passwdfile}.tmp $passwdfile || \
      ( echo "Failed to update $passwdfile - exiting now." ; rm_lock_file ; exit 1 )
done
```

Update each of the files in the group, shadow, and password directories. Make a copy of the file (i.e., cp $passwdfile ${passwdfile}.tmp), update it (i.e., echo $PASSWD_FORMAT >> ${passwdfile}.tmp), then use the mv command to put it back into place (i.e., mv ${passwdfile}.tmp $passwdfile).

The mv command makes an atomic update when moving files within the same filesystem. This means you face no risk of file corruption from the system losing power or our process getting killed. The command will either move the file into place, or it won't work at all. SAs must make file updates this way. The script will exit with an error if any part of the file-update process fails:

```
# update the UID file
NEWUID=`expr $NEXTUID + 1`
echo $NEWUID > $BASE_PATH/NEXTUID || \
 ( echo "Update of $BASE_PATH/NEXTUID failed, exiting now" ; rm_lock_file ; exit 1 )
```

Update the file used to track the next available UID:

```
# update a file used to create home dirs on the NFS server
if [ ! -f $USERNAME_FILE ]
then
        touch $USERNAME_FILE
fi

cp $USERNAME_FILE ${USERNAME_FILE}.tmp && \
echo $USERNAME >> ${USERNAME_FILE}.tmp && \
mv ${USERNAME_FILE}.tmp $USERNAME_FILE || \
 ( echo "failed to update $USERNAME_FILE with this user's account name."
    rm_lock_file ; exit 1 )
```

We store all new user accounts in a text file on the cfengine master system. We'll write another script (`PROD/repl/admin-scripts/setup_home_dirs` from the next section) that uses this file to create central home directories. The script ends with a cleanup step:

```
# if we get here without errors, clean up
rm_lock_file
```

Put this script in the previously mentioned `admin-scripts` directory, and run it from there on the *goldmaster* host when a new account is needed.

We've left one exercise for the reader: the task of removing accounts from the centralized account files. You'll probably want to use the procedure in which you edit a temporary file and `mv` it into place for that task. If the process or system crashes during an update of the account files, corrupted files could copy out during the next scheduled cfengine run. Our size minimums might catch this, but in such a scenario the corrupted files might end up being large, resulting in a successful copy and major problems.

NFS-Automounted Home Directories

We installed the host *aurora* to function as the NFS server for our future web application. We should also configure the host to export user home directories over NFS.

Configuring NFS-Mounted Home Directories

We'll configure the NFS-share export and the individual user's home directory creation with a combination of cfengine configuration and a script that's used by cfengine.

Put this line into PROD/inputs/classes/cf.main_classes:

```
homedir_server          = (     aurora )
```

Create the file PROD/inputs/hostgroups/cf.homedir_server with these contents:

```
import:
        any::
                tasks/app/nfs/cf.central_home_dirs
```

Create the file PROD/inputs/tasks/app/nfs/cf.central_home_dirs with these contents:

```
control:
        any::
                addinstallable = ( create_homedirs enable_nfs )

copy:
        homedir_server.(solaris|solarisx86)::
                $(master_etc)/USERFILE
                        dest=/export/home/USERFILE
                        mode=444
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        define=create_homedirs

                $(master_etc)/skel
                        dest=/export/home/skel
                        mode=555
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        r=inf
```

```
directories:
        homedir_server.(solaris|solarisx86)::
                /export/home  mode=755 owner=root group=root inform=false

shellcommands:
        homedir_server.create_homedirs.(solaris|solarisx86)::
                "/opt/admin-scripts/setup_home_dirs"
                        timeout=300 inform=true

        homedir_server.enable_nfs.(solaris|solarisx86)::
                "/usr/sbin/svcadm enable network/nfs/server"
                        timeout=60 inform=true

editfiles:

        homedir_server.(solaris|solarisx86)::
                { /etc/dfs/dfstab
                        AppendIfNoSuchLine  "share -F nfs -o rw,anon=0 /export/home"
                        DefineClasses           "enable_nfs"
                }
```

This should all be pretty familiar by now. The interesting part is that we sync the USERFILE file, and when it is updated we call a script that creates the needed accounts. This is the first NFS share for the host *aurora*, so we enable the NFS service when the share is added to /etc/dfs/dfstab.

Create a file at PROD/repl/admin-scripts/setup_home_dirs to create the home directories:

```
#!/bin/sh
# distributed by cfengine, don't edit locally
PATH=/usr/sbin:/usr/bin:/opt/csw/bin

USERFILE=/export/home/USERFILE

for user in `cat $USERFILE`
do
        USERDIR=/export/home/$user
        if [ ! -d $USERDIR ]
        then
                cp -r /export/home/skel $USERDIR
                chmod 750 $USERDIR
                chown -R ${user}:${user} $USERDIR
        fi
done
```

Now that the task is done, enable it in the file `PROD/inputs/hostgroups/cf.hostgroup_mappings` with this entry:

```
homedir_server::                    hostgroups/cf.homedir_server
```

Our home-directory server is ready for use by the rest of the hosts on the network.

Configuring the Automounter

Sites often utilize the automounter to mount user home directories. Instead of mounting the home NFS share from all client systems, the automounter mounts individual users' home directories on demand. After a period of no access (normally after the user is logged out for a while), the share is unmounted. Automatic share unmounting results in less maintenance, and it doesn't tax the NFS server as much. Note that most automounter packages can mount remote filesystem types other than NFS.

We're missing the `autofs` package in our base Debian installation. At this point, we add the `autofs` package to the `/srv/fai/config/package_config/FAIBASE` list of packages, so that future Debian installations have the required software. The package already exists on our Red Hat and Solaris installations.

The file names for the automounter configuration files vary slightly between Linux and Solaris. We'll create the needed configuration files and put them into our `masterfiles` repository. We created an `autofs` directory at `PROD/repl/root/etc/autofs` when we first set up our file repository in Chapter 5.

The files we'll utilize and configure on Linux are `/etc/auto.master` and `/etc/auto.home`. On Solaris, the files are `/etc/auto_master` and `/etc/auto_home`. The `auto.master` and `auto_master` files map filesystem paths to files that contain the commands to mount a remote share at that path. The `auto.home` and `auto_home` files have the actual mount commands.

Our `auto.master` and `auto_master` files each contain only a single line:

```
/home    /etc/auto.home
```

Our `auto.home` and `auto_home` files are identical, and contain only a single line:

```
*      -nolock,rsize=32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5
  aurora:/export/home/&
```

■**Note** The single line in the `auto_home` and `auto.home` files is shown as two lines due to publishing line-length limitations. It is important that you create the entry as a single line in your environment. You can download all the code for this book from the Downloads section of the Apress web site at `http://www.apress.com`.

We have a number of mount options listed, but the important thing to note is that we use a wildcard pattern on the left to match all paths requested under /home. The wildcard makes the file match /home/nate as well as /home/kirk, and look for the same path (either nate or kirk) in the share on *aurora*, using the ampersand at the end of the line.

Next, we create a task to distribute the files at PROD/inputs/tasks/os/cf.sync_autofs_ maps. This task follows what is becoming a common procedure for us, in which we define some variables to hold different file names appropriate for different hosts or operating systems, then synchronize the files, then restart the daemon(s) as appropriate:

```
control:
        any::
                addinstallable          = (       restartautofs )

                AllowRedefinitionOf     = (
                                                auto_master
                                                auto_home
                                                )

        linux::
                auto_master     = ( "auto.master" )
                auto_home       = ( "auto.home" )
                auto_net        = ( "auto.net" )
                etc_auto_home   = ( "/etc/auto.home" )
                etc_auto_master = ( "/etc/auto.master" )

        (solaris|solarisx86)::
                auto_master     = ( "auto_master" )
                auto_home       = ( "auto_home" )
                auto_net        = ( "auto_net" )
                etc_auto_home   = ( "/etc/auto_home" )
                etc_auto_master = ( "/etc/auto_master" )

copy:
        any::
                $(master_etc)/autofs/$(auto_master)
                        dest=$(etc_auto_master)
                        mode=444
                        owner=root
                        group=root
                        server=$(fileserver)
                        trustkey=true
```

```
                            type=checksum
                            encrypt=true
                            define=restartautofs

                  $(master_etc)/autofs/$(auto_home)
                            dest=$(etc_auto_home)
                            mode=444
                            owner=root
                            group=root
                            server=$(fileserver)
                            trustkey=true
                            type=checksum
                            encrypt=true
                            define=restartautofs

shellcommands:
        (debian|redhat).restartautofs::
                # when config is updated, restart autofs
                "/etc/init.d/autofs reload"
                        timeout=60 inform=true

        (solaris|solarisx86).restartautofs::
                # when config is updated, restart autofs
                "/usr/sbin/svcadm restart autofs"
                        timeout=180 inform=false

processes:
        debian|redhat::
                "automount" restart "/etc/init.d/autofs start" inform=true

    solaris|solarisx86::
        "/usr/sbin/svcadm enable autofs ; /usr/sbin/svcadm restart autofs" inform=true
```

We start the automounter when the process isn't found in the process list. We attempt to enable the NFS service on Solaris when it's not running, then we try to restart it. We don't know what the problem is when it's not running on Solaris, so the enable step seems like a logical solution to one possible cause.

Import this task into PROD/inputs/hostgroups/cf.any to give all your hosts a working automounter configuration.

We now have a system to add users, and we also have a shared home-directory server. This should suffice until you can implement a network-enabled authentication scheme later.

Routing Mail

Mail is the primary message-passing mechanism at UNIX-based sites. You use mail to notify users of cron-job output, `cfexecd` sends `cfagent` output via e-mail, and many application developers and SAs utilize e-mail to send information directly from applications and scripts.

Mail relays on internal networks route e-mail and queue it up for the rest of the hosts on the network when remote destinations become unreachable. You should centralize disk space and CPU resources needed for mail queuing and processing. In addition, it's simpler to configure a centralized set of mail relays to handle special mail-routing tables and aliases than it is to configure all the mail-transfer agents on all machines at a site.

We'll use our *etchlamp* Debian host as our site's mail relay. We've built this host entirely using automation, so it's the sensible place to continue to focus infrastructure services.

We add a CNAME for *relayhost.campin.net* to `PROD/repl/root/etc/bind/debian-ext/db.campin.net`, and it'll simply go out to *etchlamp* on the next `cfexecd` run:

```
relayhost       IN      CNAME   etchlamp
```

Be sure to increment the serial number in the zone file.

We run postfix on all our Debian hosts, and we'll stick with postfix as our mail-relay Mail Transfer Agent (MTA). The default postfix configuration on *etchlamp* needs some modifications from the original file placed in `/etc/postfix/main.cf`. Modify the file like this:

```
smtpd_banner = $myhostname ESMTP $mail_name (Debian/GNU)
biff = no

# appending .domain is the MUA's job.
append_dot_mydomain = no

# TLS parameters
smtpd_tls_cert_file=/etc/ssl/certs/ssl-cert-snakeoil.pem
smtpd_tls_key_file=/etc/ssl/private/ssl-cert-snakeoil.key
smtpd_use_tls=yes
smtpd_tls_session_cache_database = btree:${queue_directory}/smtpd_scache
smtp_tls_session_cache_database = btree:${queue_directory}/smtp_scache

myhostname = campin.net
alias_maps = hash:/etc/aliases
alias_database = hash:/etc/aliases
mydestination = campin.net
```

```
myorigin = campin.net
mynetworks = 127.0.0.0/8, 192.168.1.0/24
mailbox_command = procmail -a "$EXTENSION"
mailbox_size_limit = 0
recipient_delimiter = +
inet_interfaces = all
virtual_maps = hash:/etc/postfix/virtual
```

Next, create a file that we'll copy to /etc/postfix/virtual on the mail relay:

```
campin.net              OK
@campin.net             sysadmins@foo.bar
```

We use the virtual-domain functionality of postfix to alias the entire *campin.net* domain to one e-mail address: sysadmins@foo.bar. This ensures that any mail sent will arrive in the SA team's mailbox (hosted with an e-mail–hosting provider). Later, we can use the same virtual table to forward specific e-mail addresses to other destinations, instead of the single catch-all address we're using now.

When the source file /etc/postfix/virtual is updated, we need to run this command as root:

```
# /usr/sbin/postmap /etc/postfix/virtual
```

This builds a new /etc/postfix/virtual.db file, which is what postfix actually uses. We'll configure cfengine to perform that step for us automatically.

Place the two files in a replication directory on the cfengine master (*goldmaster*), and also create a new directory under the tasks hierarchy intended for postfix:

```
# mkdir /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/postfix/
# cp main.cf virtual /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/postfix/
# mkdir /var/lib/cfengine2/masterfiles/PROD/inputs/tasks/app/postfix
```

First, create a class called relayhost, and place the host *etchlamp* in it. Place this line in PROD/inputs/classes/cf.main_classes:

```
relayhost               = (    etchlamp )
```

Now create the task PROD/inputs/tasks/app/cf.sync_postfix_config with these contents:

```
control:
        debian_4_0.relayhost::
                main_cf      = ( "main.cf_debian-relayhost" )
                virtual        = ( "virtual-relayhost" )


copy:
        debian_4_0.relayhost::
                $(master_etc)/postfix/$(main_cf)
                                dest=/etc/postfix/main.cf
                                mode=444
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                # we already have reloadpostfix from
                                # tasks/os/cf.resolve_conf, we are reusing it
                                define=reloadpostfix

                $(master_etc)/postfix/$(virtual)
                                dest=/etc/postfix/virtual
                                mode=444
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=rebuild_virtual_map
```

We define variables for the virtual and main.cf files, and copy them individually. They're set up individually because different actions are required when the files are updated. We are careful to copy the configuration files that we've prepared only to Debian 4.0, using the debian_4_0 class. When Debian 5.0 ("Lenny") is released, we'll have to test our config files against the postfix version that it uses. We might have to develop a new "relayhost" postfix configuration file specifically for Lenny when we upgrade or reimage the "relayhost" system to use the newer Debian version. Once again, we assume that something won't work until we can prove that it will.

Here we use the copy action to rebuild the virtual map when it is updated:

```
shellcommands:
        rebuild_virtual_map::
                "/usr/sbin/postmap /etc/postfix/virtual ; /usr/sbin/postfix reload "
                        timeout=60 inform=true
```

Now we need another hostgroup file for the "relayhost" role. We create `PROD/inputs/hostgroups/cf.relayhost` with these contents:

```
import:
        any::
                tasks/app/postfix/cf.sync_postfix_config
```

Then to finish the job, map the new class to the hostgroup file by adding this line to `PROD/inputs/hostgroups/cf.hostgroup_mappings`:

```
relayhost::                        hostgroups/cf.relayhost
```

Now *etchlamp* is properly set up as our mail-relay host. When our network is larger, we can simply add another Debian 4.0 host to the `relayhost` class in `PROD/inputs/control/cf.main_classes`, thus properly configuring it as another mail relay. Then we just update the DNS to have two A records for *relayhost.campin.net*, so that the load is shared between the two. An additional benefit of having two hosts serving in the "relayhost" system role is that if one host fails, mail will still make it off our end systems.

You have several options to accomplish the task of configuring systems across the site to utilize the mail relay. For example, you can configure Sendmail, qmail, and postfix in a "nullclient" configuration where they blindly forward all mail off the local system. Or you could use the local aliases file to forward mail as well. The method, and automation of that method, is left up to the reader. You should now have a solid understanding of how to use cfengine to automate these configuration changes once you've worked out the procedure on one or more test systems.

Looking Back

In a rather short amount of time, we've gone from having no systems at all to having a basic UNIX/Linux infrastructure up and running. This by itself might not be very interesting, but what is noteworthy is that everything we've done to set up our infrastructure was accomplished using automation.

If our DNS server (and mail-relay) host suffers a hard-drive crash, we will simply replace the drive and reimage the host using FAI and the original hostname. Cfengine will configure a fully functional replacement system automatically, with no intervention required by the SA staff. The benefits of this are obvious:

- The risk of errors introduced during configuration of the replacement host is reduced to zero (or near zero). Any errors would be the result of further hardware issues.

- The addition of new hosts to share the load of existing services is equally trivial: you need only to add additional hosts to the role-based classes in cfengine, and cfengine will configure the new host properly for you. From that point, the only steps are to update DNS records or configure applications to use the additional host(s).

- The difficulty of training new SA staff is reduced. The applications in use at your site, along with the configurations used, are centralized in cfengine. The new SAs can simply read the cfengine and application-configuration files to get a complete picture of how things run at your site.

We now have sufficient core services in place at our site to support customer-facing applications. In the next chapter, we'll take advantage of that fact, and deploy a web site.

Deploying Your First Application

The first application in our new environment is a web site, the *campin.net* shopping web site mentioned in earlier chapters. Our company is going to launch a PHP-based web site where customers can purchase camping equipment. In keeping with our focus on automation, we provide only basic information about the services and protocols that we configure. We will refer you to sources of in-depth information as appropriate.

Deploying and Configuring the Apache Web Server

The Apache web server is the reference implementation of the HTTP protocol, and it has been the most widely deployed web server on the Internet since 1996. It is an open source project, and it is included or available with most Linux distributions. See `http://httpd.apache.org/ABOUT_APACHE.html` for more information.

Apache is relatively easy to configure, and it supports all common languages that web developers need. We'll use it to host our web site.

The Apache Package from Red Hat

Back in Chapter 6, we imaged the system *rhlamp* with the packages it needed to function as a web server. We did this by selecting the check box for "Web Server" when we selected server packages from within the Kickstart Configurator application.

The default installation of Apache on Red Hat Enterprise Linux 5.2 is version 2.2.3. Utilizing the Red Hat package means you won't have to manually build and redeploy when security and bug-fix releases for Apache become available. As long as Red Hat still supports our system, we can simply install the updated package from Red Hat.

By default, Red Hat's Apache package supports PHP, and it configures an empty directory that's ready to be populated with content. This is the directory `/var/www/html`. Red Hat provides a fully functional web server upon installation of the package, and you'll have little reason to redo all the work that the kind folks at Red Hat have done for us.

Many experienced SAs like to build Apache from source on their own, often for performance reasons. Performance tuning isn't needed for most web sites at the early stages,

and most of the tuning is done with configuration directives rather than build options. This means many sites don't need to use another Apache package or build their own.

We will configure Red Hat Apache for our *http://shop.campin.net* web site. The configuration files for the Red Hat Apache package reside in the /etc/httpd directory. You'll find several directories and files inside that directory:

```
# cd /etc/httpd/
# ls
./  ../  conf/  conf.d/  logs@  modules@  run@
# ls conf
./  ../  httpd.conf  magic
# ls conf.d
./   manual.conf  php.conf       python.conf  squid.conf  webalizer.conf
../  perl.conf    proxy_ajp.conf  README      ssl.conf    welcome.conf
```

Inside the conf.d directory, all the files are processed in alphabetical order. Until we do something to change it, the absence of any files in the /var/www/html directory causes Apache to serve a default page with the text "Red Hat Enterprise Linux Test Page" displayed prominently at the top.

To have Apache serve our own content, we simply have to put our web content into the /var/www/html directory. You can do this in cfengine with a simple file copy. Edit the file /etc/httpd/conf/httpd.conf on *rhlamp* and change the line:

```
#ServerName www.example.com:80
```

to this:

```
ServerName shop.campin.net:80
```

We'll have a load balancer on a public IP, which forwards traffic to *rhlamp* on port 80 in order to serve our web site. If we require additional web servers later on, we'll image more Red Hat web servers and simply add the additional hosts into the load-balancer configuration. We'll make an entry in the public DNS when we're ready, and the IP will be on the load balancer, not on any of our web-server hosts. We won't cover that in this book, however. For more information on load balancing, see http://en.wikipedia.org/wiki/Load_balancing_(computing).

Save the httpd.conf file with the modified ServerName directive on the cfengine master at the location PROD/repl/root/etc/httpd/conf/httpd.conf. Next, create a class for the web servers at our site (with only one member for now) by adding this line to PROD/inputs/classes/cf.main_classes:

```
web_server             = (     rhlamp )
```

Create a new directory for Apache in the PROD/inputs/tasks/app directory:

```
# mkdir /var/lib/cfengine2/masterfiles/PROD/inputs/tasks/app/apache
```

Put these contents in the task PROD/inputs/tasks/app/apache/cf.sync_httpd_conf:

```
control:
        any::
                AddInstallable = ( reload_httpd )

copy:
        redhat.web_server::
                $(master_etc)/httpd/conf/httpd.conf
                                dest=/etc/httpd/conf/httpd.conf
                                mode=444
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=reload_httpd

shellcommands:
        redhat.web_server.reload_httpd::
                "/etc/init.d/httpd reload"
                        timeout=60 inform=true
```

We stick with the same Apache DocumentRoot, and everything else configured in the /etc/httpd/conf.d directory. We wish to change only the ServerName parameter at this point.

Create a hostgroup file at the location PROD/inputs/hostgroups/cf.web_server:

```
import:
        any::
                tasks/app/apache/cf.sync_httpd_conf
```

Then activate it as usual in the PROD/inputs/hostgroups/cf.hostgroup_mappings file with this line:

```
web_server::                            hostgroups/cf.web_server
```

We can add web content to /var/www/html at any time, but we'll hold off on that until after we look at building and deploying Apache ourselves.

The final step is to make sure that Apache is running on your web servers. Create the task `PROD/inputs/tasks/app/apache/cf.apache_start` with these contents:

```
processes:
      web_server.redhat::
              "httpd" restart "/etc/init.d/httpd start" inform=false umask=022
```

Then place this line into `PROD/inputs/hostgroups/cf.web_server`:

```
tasks/app/apache/cf.apache_start
```

The `cf.apache_start` file is easy to modify if we decide to run a different Apache, i.e., one that we build and deploy ourselves.

Building Apache from Source

There is one very good reason to build your own Apache and PHP from the initial stage: a security problem or new feature that your web developers need might require a newer version of Apache or PHP than the ones bundled with Red Hat 5.2. If you install Apache and PHP from source from the start, you won't have to learn to build and deploy Apache on a rushed schedule later. We'll install Apache and PHP to version-specific directories—the same way we deployed cfengine on Red Hat during the Kickstart installation process—and use a symlink to maintain a consistent path to our Apache installation.

We'll keep the configuration files and web content in a directory separate from the Apache program binaries, which will simplify later upgrades. Check with your site's web developers to see which Apache and PHP options are required, and enable only the needed functionality. Doing this will limit your exposure to potential security issues. We'll demonstrate how to build a basic PHP-enabled Apache server. You'll need to have C development packages installed, such as `gcc`, `make`, and so on. In Chapter 5, we installed these on the *rhmaster* system in order to compile cfengine.

First we'll download the latest Apache and PHP sources, place them in `/var/tmp`, and extract the tarballs:

```
# tar zxf httpd-2.2.9.tar.gz
# tar zxf php-5.2.6.tar.gz
# cd httpd-2.2.9
```

Next we build Apache, making sure to enable shared modules with the "enable-so" option:

```
# ./configure --prefix=/usr/pkg/httpd-2.2.9 --enable-so --with-mpm=prefork
# make -j2 && make install
```

Use these commands to test whether the binary works:

```
# /usr/pkg/httpd-2.2.9/bin/httpd -v
Server version: Apache/2.2.9 (Unix)
Server built:   Aug  4 2008 23:53:23
# /usr/pkg/httpd-2.2.9/bin/apachectl start
```

You should be able to access the web server at *http://rhmaster/* and see the text, "It works!" Now shut down the web server with this command:

```
# /usr/pkg/httpd-2.2.9/bin/apachectl stop
```

We're done with Apache for now. Next, we build PHP:

```
# cd ../php-5.2.6
# ./configure --prefix=/usr/pkg/php-5.2.6 --with-apxs2=/usr/pkg/httpd-2.2.9/bin/apxs
# make
# make test
# make install
# cp php.ini-dist /usr/pkg/php-5.2.6/lib/php/php.ini
```

Edit the `php.ini` file to suit your site's needs (refer to `http://www.php.net/manual/en/ini.php` for assistance). Next, we need to enable PHP in the Apache configuration file. Verify that `/usr/pkg/httpd-2.2.9/conf/httpd.conf` has this line:

```
LoadModule php5_module modules/libphp5.so
```

Now you need to configure Apache to treat certain file types as PHP files and invoke the PHP module to process and serve them. You can do this by adding this line to `httpd.conf`:

```
AddType application/x-httpd-php .php .phtml
```

Afterward, start Apache with this line:

```
# /usr/pkg/httpd-2.2.9/bin/apachectl start
```

Once Apache is running again, create a file named `/usr/pkg/httpd-2.2.9/htdocs/index.php` with these contents:

```
<html>
<head>
<title>PHP Test</title>
</head>
```

```
<body>
<?php phpinfo(); ?>
</body>
</html>
```

Use this `chmod` command to make sure the file is publicly readable:

```
# chmod a+r index.php
```

Visit your new web server in a web browser at the URL *http://rhmaster/index.php* (substitute your host's hostname as applicable); you should see a page with the PHP logo, the PHP version you've compiled, and HTML tables filled with information on your PHP installation.

Shut down Apache because we won't use it again on this system. You should also remove the file with the `phpinfo()` call in it because it gives away information that might be useful to attackers:

```
# /usr/pkg/httpd-2.2.9/bin/apachectl start
# rm usr/pkg/httpd-2.2.9/htdocs/index.php
```

We now have an Apache build ready to be deployed to Red Hat 5.2 (32-bit, x86-based) systems. We'll want to distribute the binaries using one of several options: rsync, cfengine, Network File System (NFS), or perhaps even Subversion.

If you encounter issues building Apache or PHP, see the INSTALL file included with the source distribution for each. It contains a wealth of information on installation and configuration, as does each of their respective web sites: `http://httpd.apache.org` and `http://www.php.net`.

Sharing Data Between Systems

Our web developers have created a large collection of web content, which should work on any Apache server running PHP 5. We'll explore several ways to distribute the content, and discuss the benefits of each method.

Synchronizing Data with rsync

Rsync is a wonderful program that allows you to transfer files and directories from one host to another. It might not sound very impressive; you are probably thinking you can do all of this yourself with `scp` or cfengine. You *could*—rsync, in fact, can use the SSH protocol to do its work—but you probably couldn't do it as well as rsync does.

What rsync adds to the mix is the ability to efficiently mirror files between two hosts. The files are compared using their timestamps or checksums, and only the necessary files

are transferred. In fact, when files do change, only the portions that have changed are sent over the network. This makes rsync very efficient. We regularly use rsync to synchronize about 1GB of data; it takes only a couple seconds when there are no (or few) changes.

You can also use rsync to delete files and modify permissions to achieve an exact copy. It can copy device files, create symbolic links, preserve ownership, and synchronize timestamps. Rsync also supports both include and exclude patterns that allow you to specify exactly which files you want synchronized.

■**Note** It is important to remember that the rsync program must be installed on both the remote and local systems to synchronize data between those systems.

Possible Uses of rsync

You'll find rsync very useful for synchronizing applications and their data. You could, for example, use rsync to synchronize /usr/local/ across several systems. In most environments, /usr/local/ doesn't change often, so using rsync once per day can prove significantly more efficient than using network filesystems. In addition, using rsync reduces network use and improves application access time. However, not surprisingly, it uses much more disk space because each system has its own complete copy of the application directory.

Clusters of web servers have become commonplace in the modern web-enabled world. You might have anywhere from 200 to 500 web servers behind a set of load balancers. Each of these web servers might need to access a large amount of web content. Creating an efficient and reliable network filesystem to store the web-server software and content can be expensive, but because hard drives are so inexpensive these days, each system can simply store all its content on its local drive and update it daily with rsync. For quicker updates, you can even combine a pull and push method—the servers can check for changes daily (the pull portion), and you could also push new content on demand to update the servers quickly.

One potential drawback to using rsync is that it has real trouble showing any respect for changes made on the system being updated. This means that any files that were locally modified will be replaced with the copy from the server, and any deleted files will be added again. Also, if you use the --delete switch, any new files will be erased. This means you must make *all* changes on the master server and never make them on the client systems. You should properly train anybody who needs to make changes if you don't want them to learn the lesson the hard way.

Some would consider this potential drawback a benefit—when you have ten copies of what is supposed to be identical data, the last thing you want is people making changes in one place and forgetting to propagate that change to other systems. Another benefit

is that you can add debugging; yet another, that you can try something on a system and erase all the temporary changes by resyncing the data.

You can also tell rsync to dereference symbolic links when copying (using the `-L` switch). If you do, rsync creates a copy of any symbolic link in the source directory as a regular file (or directory) in the destination directory. The destination tree will take up more space as a result, but eliminating symbolic links can be useful in some situations. If, for example, you need to replicate data regularly, but the application using that data does not work with symbolic links, you can use rsync to solve your problems. You can also use symbolic links to make managing the original copy easier and then use rsync to make a useable copy (you can even make the copy from and to the same system).

Some examples in which dereferencing symbolic links might be useful are `chroot` environments, anonymous FTP sites, and directories that are exported via NFS. Although symbolic links work fine within the anonymous FTP directory, for example, they cannot reference files below the root of the anonymous directory. If you were to use rsync to transfer some or all of your FTP content into your anonymous FTP directory, however, any symbolic links would be dereferenced during that process (if the `-L` switch is used).

Deciding Which rsync Transport Protocol to Use

We are pleased to report, in this book's second edition, that rsync now uses the SSH protocol by default to communicate with a remote system, allowing it to use all the standard SSH authentication methods and perform encrypted file transfers. We recommend using SSH if at all possible. You can tell rsync to use RSH by setting the `RSYNC_RSH` environment variable to the location of your `rsh` binary (such as `/usr/bin/rsh`), but this would be unwise on all but the most private and secure networks.

If you want to include extra options for the SSH program, you can specify them with the `-e` argument to rsync:

```
# rsync -av -e "ssh -2 -o StrictHostKeyChecking=no -o CheckHostIP=no -o \
ForwardAgent=no -i ~/.ssh/id_dsa_report_sync" /tmp/reports/ aurora:~/reports
```

The `-2` option specifies that protocol version 2 should be used for this encrypted connection. The `-i` option provides a special location for your private-key file. Agent forwarding is not necessary for file transfers, so you can turn off the `ForwardAgent` option.

With the options `CheckHostIP` and `StrictHostKeyChecking` disabled, `ssh` never prompts you for a host's public key. This is fine, in some cases. For instance, if the information you are pushing is not sensitive, it doesn't matter if you push it to an attacker's system. If you are pulling from a server, however, you might not want these options because somebody could spoof your server and push out their own data to your systems.

We should mention that all the preceding SSH client settings could be set in the user's `~/.ssh/config` file, with the configuration based on the remote host's name. The

reason a user might choose to use the command line instead is that they don't want those options applied to *all* SSH connections to that particular host.

Basic Use of rsync

By our estimate, rsync offers approximately 50 command-line options. Rather than cover each option, we'll show you some useful examples. When trying out these commands, you might find it useful to use the verbose switch (`-v`) and/or the dry-run switch (`-n`). You should also be careful when trying out rsync because you could accidentally copy large amounts of data to places it does not need to be. The dry-run switch can help with the debugging process because it will prevent rsync from transferring any files. So you can see what would have happened without actually making any file modifications.

Here is a simple example that copies /usr/local/ to a remote system (recursively, in archive mode, using the -a switch):

```
$ rsync -a /usr/local root@remote_host:/usr/local/
```

If you were to run this command, you would see that it doesn't quite work as expected. The /usr/local/bin/ directory on the local system is pushed as /usr/local/local/bin/ on the remote system. The reason this happens is that we forgot the slash on the end of the source directory. We asked rsync to copy the file (or directory) /usr/local into the remote directory /usr/local/.

Any of the following commands will work the way you might have expected the previous command to operate. Personally, we prefer to use the first one listed here because we think it is the most readable. We would not recommend running any of these other commands unless you really want to copy all of /usr/local/ to another system (just in case you didn't learn your lesson when you tried the last example):

```
$ rsync -a /usr/local/ root@remote_host:/usr/local/
$ rsync -a /usr/local/ root@remote_host:/usr/local
$ rsync -a /usr/local root@remote_host:/usr/
$ rsync -a /usr/local root@remote_host:/usr
```

The -a switch tells rsync to operate in archive mode. This causes it to operate recursively and preserve permissions, ownership, symbolic links, device files, access times, and so on. In fact, using the -a switch works the same as using the options -rlptgoD.

By default, rsync leaves a file alone if it has exactly the same timestamp and size as the source file. You must specify the -a switch or the -t switch to preserve file timestamps. If you do not use either of these options, every file will be transmitted every time you execute rsync. This is fine if you are using rsync for a one-time transfer, but it is not very efficient when you run rsync multiple times.

If you previously used a different method to transfer files and you are not sure the timestamps are correct, but you know that most of the files are already identical, you can use the `--size-only` option along with the `-a` switch for the first transfer. This option causes rsync to ignore the timestamps and use only the file sizes to find differences. It then sets all timestamps properly (because of the `-a` switch) so that future runs on the same file do not need the `--size-only` switch. For a small amount of files this doesn't matter, but if you are transferring lots of data or using a slow connection, this can make the first synchronization less painful.

The only problem with this example is that it does not delete any files on the remote system. Suppose you used to have `/usr/local/program-2.0/`, but now you have `/usr/local/program-3.0/`. You might have deleted `/usr/local/program-2.0/` on the server, but it has not been deleted on the other systems.

This is where the `--delete` switch becomes useful. This switch tells rsync to delete any files in the destination directory that are not in the source directory. You must be very careful with the `--delete` switch because it has the potential to do major damage. You could, for example, accidentally run this command:

```
$ rsync -a --delete /tmp/ root@remote_host:/
```

which would wipe out everything on the entire remote host and replace it with the contents of the local `/tmp/` directory. We have a feeling this is not something you would like to do for fun (in fact, we feel a bit guilty for including it as an example in the book—please let us know if you use that command to accidentally destroy one of your systems!). However, as long as you are careful when you use this option, you will find that it can be very useful. As a minimal safeguard, rsync will never delete files if any errors occur during the synchronization process.

You could expand the preceding example to remove files that shouldn't be on the destination system. This allows you to create an exact copy of the source files on the destination system. Here's the command you would run to accomplish this:

```
$ rsync -a --delete /usr/local/ root@remote_host:/usr/local/
```

After this command runs, the remote system will have exactly the same files in its `/usr/local/` directory as the local system—no less, and no more.

Don't forget that rsync works equally well when the source and destination are on the same system. If you're about to upgrade an important binary-only, third-party software package in the directory `/usr/pkg/FooSoft`, you can make a perfect copy of the old installation like this:

```
$ rsync -a /usr/pkg/FooSoft/ /usr/pkg/FooSoft.bak/
```

This allows you to attempt the upgrade with less risk. If anything goes wrong, simply move the `FooSoft` directory to `FooSoft.bad`, and move `FooSoft.bak` to `FooSoft`. As long as

your original copy operation succeeded (you had the required privileges to read and write the files, and there was adequate disk space), then this fallback plan will enable you to attempt many risky operations with a safety net in place.

Synchronizing Web Content with rsync and cfengine

In the previous section, you saw how to push out `/usr/local/` to another system. You could easily extend this practice to pull your web content to the necessary system. Performing the synchronization is actually quite easy, but automating the procedure in a secure manner is another matter. Cfengine handles two-way authentication for us, and in this book we always use encryption in our cfengine copies. In addition to those benefits, cfengine allows you to define classes based on changes made, which allow you to run further actions, such as restarting Apache when the configuration files are updated. We don't, however, need to restart Apache when the web content is updated, so this should be a good and simple problem for us to solve with rsync.

We'll set up a dedicated rsync server on our cfengine master. A major reason for choosing to use an rsync server is that we don't want to use system accounts for access. You could use forced commands with SSH public-key authentication to force secure authentication and limited system access, but that will unnecessarily complicate this section. We'll keep it as simple as possible.

As always, we automate this process. Even though we're not using cfengine for this copy, we will use it to set up and run rsync. We won't ever have to wonder how the rsync daemon was set up or how the client is run; it will be thoroughly documented in our cfengine configuration files for later reference or for application to another system.

■**Note** We are running Debian GNU/Linux on our cfengine master, so the examples work properly only on Debian. Modifying this procedure to work on Red Hat Linux or Solaris wouldn't be very difficult, but it would complicate the examples unnecessarily. We hope that you learn enough about the procedure from this example to get it working on your own systems.

Place the web content in a directory in your master cfengine replication tree. We'll use the directory `PROD/repl/root/var/www/html`. Create a new class for the role of the web-content master by adding these lines in `PROD/inputs/classes/cf.main_classes`:

```
# when this host changes, update the web_master variable in cf.control_cfagent_conf
web_master              = (     goldmaster )
```

Next, add these lines to create a variable in `PROD/inputs/control/cf.control_cfagent_conf`:

```
# when this changes, update the web_master class in cf.main_classes
web_master      = ( goldmaster.campin.net )
```

We use both the variable and the class to abstract the hostname away from the role name. You could use a DNS alias here instead of the variable, but the variable is really just another way to create an alias. In this case, we keep the alias creation contained to cfengine itself. We added those comments in the files as reminders to ourselves about the two places in cfengine where we need to keep the role name synchronized.

Next, place a file at `PROD/repl/root/etc/rsync/rsyncd.conf-www` with these contents:

```
[www-content]
      comment = archives
      path = /var/lib/cfengine2/masterfiles/PROD/repl/root/var/www/html
      use chroot = no
      max connections = 5
      lock file = /var/tmp/rsyncd.lock
      read only = yes
      list = no
      hosts allow = rhlamp*.campin.net
      hosts deny = 0.0.0.0/0
      timeout = 600
      refuse options = delete delete-excluded delete-after ignore-errors \
                              max-delete partial force
      dont compress = *
```

We'll use this rsync configuration file to share our master web-repository directory to rsync clients. We'll create a task for starting an rsync server that utilizes this configuration file, in the task `PROD/inputs/tasks/app/rsync/cf.enable_rsync_daemon`. Create that file now with these contents (you'll need to create the rsync directory where it resides):

```
classes:  # synonym groups:
      have_usr_bin_rsync     = ( "/usr/bin/test -f /usr/bin/rsync " )
      have_etc_inetd_conf    = ( "/usr/bin/test -f /etc/inetd.conf " )

control:
      any::
              AddInstallable          = ( hup_inetd  )

      web_master.debian::
              rsyncd_conf     = ( "rsyncd.conf-www" )
```

```
copy:
        web_master.debian::
                $(master_etc)/rsync/rsyncd.conf-www
                                dest=/etc/rsyncd.conf
                                mode=444
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true


editfiles:
        web_master.debian.have_etc_inetd_conf.have_usr_bin_rsync::
                { /etc/inetd.conf
                        AppendIfNoSuchLine "rsync stream tcp nowait daemon ➥
/usr/bin/rsync rsyncd --daemon --config=/etc/rsyncd.conf"
                        DefineClasses          "hup_inetd"
                }

processes:
        web_master.debian.hup_inetd::
                "inetd" signal=hup inform=true
```

■**Caution** You'll see the code-continuation character (➥) in the `AppendIfNoSuchLine` entry in the pre-
ceding `editfiles` section. This character signifies that the line in which it appears is actually a single line,
but could not be represented as such because of print-publishing restrictions. It is important that you create
the entry as a single line in your environment. You can download all the code for this book from the Down-
loads section of the Apress web site at `http://www.apress.com`.

Now add a hostgroup for the web master, a file at `PROD/inputs/hostgroups/cf.web_mas-`
`ter` with the contents:

```
import:
        any::
                tasks/app/rsync/cf.enable_rsync_daemon
```

To finish up, add a line to the hostgroup mappings file (`PROD/inputs/hostgroups/cf.hostgroup_mappings`) like this:

```
web_master::                        hostgroups/cf.web_master
```

If you don't observe updates being made to `/etc/inetd.conf`, check whether rsync is installed. Feel free to install it using `apt-get`, but make sure all future Debian installations have rsync installed by adding the line `rsync` to the FAI base packages list on *goldmaster* at `/srv/fai/config/package_config/FAIBASE`. In our case, we already have it there.

Now we can synchronize over rsync from a client in a read-only manner. We can use a simple `shellcommands` section from a cfengine task to perform the synchronization. Create a task at `PROD/inputs/tasks/app/apache/cf.sync_web_content` with these contents:

```
directories:
      web_server::
              /var/www/html  mode=755 owner=root group=root inform=true


shellcommands:
      web_server.Min00_05::
        "/usr/bin/rsync -aq --delete ${web_master}\:\:www-content/ /var/www/html"
                      timeout=600 inform=false umask=022
```

The preceding `shellcommands` action will result in a copy of the `/var/www/html` directory, using rsync, at the top of the hour. Remember that we run `cfagent` (from `cfexecd`) at the top of the hour, 20 minutes after the hour, and 40 minutes after the hour. Because we use a 5-minute `SplayTime`, we need to specify the 5-minute range between minute 00 and minute 05. We also set the umask to be less restrictive than the cfengine default, so that the Apache process can read the files that are copied. The rsync `-a` flag should take care of the file permissions for us, but we are extremely cautious.

Note that it's necessary to escape the colons in the rsync command because cfengine interprets a colon as a list-iteration operator. If you're curious, see `http://www.cfengine.org/docs/cfengine-Reference.html#Iteration-over-lists` for more information on using list variables in cfengine. We avoid utilizing that feature in this book for simplicity.

Activate the task with this line in `PROD/inputs/hostgroups/cf.web_server`:

```
tasks/app/apache/cf.sync_web_content
```

Now all we have to do to update our web site is to update the master web repository on the cfengine master. If we add more web servers, we simply add them to the `web_server` group in `PROD/inputs/classes/cf.main_classes`, and all the appropriate tasks will execute on the host and bring it up to date with our Apache configuration and web content.

Synchronizing Apache and PHP with rsync

To use rsync to synchronize the Apache and PHP binaries that we built, we can use the same method we used for synchronizing the web content. We'll need additional steps here that the web-content copy didn't need, such as handling Apache configuration and startup.

If you're following along with the examples in the book and you want to utilize an Apache server that you built from source, you must disable the Apache startup script and the cfengine task to start it up when it isn't running. This means you should comment out this line in `PROD/inputs/hostgroups/cf.web_server`:

```
tasks/app/apache/cf.sync_httpd_conf
```

You'll also want to run this manually on your Red Hat web server before doing any more automation:

```
# chkconfig httpd off
```

Here's the reason we do this manually: that daemon wasn't enabled upon installation; we previously turned it on. Turning it off now restores the system to its default state. We have a strong feeling that we'll stick with the Apache that we compiled ourselves, so we'll act as if we never turned on the Red Hat version of Apache in the first place.

Let's add three new sections for Apache to our "web master" role's rsync daemon configuration file at `PROD/repl/root/etc/rsync/rsyncd.conf-www`:

```
[apache-2.2.9]
        comment = Apache 2.2.9
        path = /var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/httpd-2.2.9
        use chroot = no
        max connections = 40
        lock file = /var/tmp/rsyncd2.lock
        read only = yes
        list = no
        hosts allow = rhlamp*.campin.net
        hosts deny = 0.0.0.0/0
        #transfer logging = yes
        #log format = %t: host %h (%a) %o %f (%l bytes). Total %b bytes.
        timeout = 3600
        refuse options = delete delete-excluded delete-after ignore-errors \
                                    max-delete partial force
        dont compress = *
```

```
[php-5.2.6]
        comment = PHP 5.2.6
        path = /var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/php-5.2.6
        use chroot = no
        max connections = 40
        lock file = /var/tmp/rsyncd3.lock
        read only = yes
        list = no
        hosts allow = rhlamp*.campin.net
        hosts deny = 0.0.0.0/0
        #transfer logging = yes
        #log format = %t: host %h (%a) %o %f (%l bytes). Total %b bytes.
        timeout = 3600
        refuse options = delete delete-excluded delete-after ignore-errors \
                                    max-delete partial force
        dont compress = *


[httpd-conf]
        comment = Apache configuration files
        path = /var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/httpd-conf
        use chroot = no
        max connections = 40
        lock file = /var/tmp/rsyncd4.lock
        read only = yes
        list = no
        hosts allow = rhlamp*.campin.net
        hosts deny = 0.0.0.0/0
        #transfer logging = yes
        #log format = %t: host %h (%a) %o %f (%l bytes). Total %b bytes.
        timeout = 3600
        refuse options = delete delete-excluded delete-after ignore-errors \
                                    max-delete partial force
        dont compress = *
```

Now create the version-specific Apache and PHP master directories on the "web master" system (*goldmaster*, the cfengine master):

```
# mkdir -p /var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/httpd-2.2.9
# mkdir -p /var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/php-5.2.6
```

Now we'll rsync our Apache and PHP binaries over from the *rhmaster* system where we built it. Run this command on the host *goldmaster*:

```
# rsync -avze ssh --progress --partial rhmaster:/usr/pkg/httpd-2.2.9/ \
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/httpd-2.2.9
receiving file list ...
1196 files to consider
./
bin/
bin/ab
        67543 100%  381.27kB/s    0:00:00 (xfer#1, to-check=1193/1196)
bin/apachectl
         3435 100%   18.53kB/s    0:00:00 (xfer#2, to-check=1192/1196)
bin/apr-1-config
         6974 100%   25.80kB/s    0:00:00 (xfer#3, to-check=1191/1196)
bin/apu-1-config
         6205 100%   22.12kB/s    0:00:00 (xfer#4, to-check=1190/1196)
bin/apxs
        22599 100%   76.63kB/s    0:00:00 (xfer#5, to-check=1189/1196)
bin/checkgid
         9554 100%   31.31kB/s    0:00:00 (xfer#6, to-check=1188/1196)
bin/dbmmanage
         8876 100%   28.05kB/s    0:00:00 (xfer#7, to-check=1187/1196)
bin/envvars
          980 100%    2.99kB/s    0:00:00 (xfer#8, to-check=1186/1196)
bin/envvars-std
          980 100%    2.87kB/s    0:00:00 (xfer#9, to-check=1185/1196)
```

(Note: we've truncated the output for simplicity.)
The options that we gave are:

- The -a flag designates archive-copy mode.

- The -v flag provides increased verbosity about what rsync is doing. We won't supply this argument in our automated rsync copies.

- The -z flag, for compression, helps on low-bandwidth or heavily utilized network links. The extra CPU overhead probably increases the transfer time on a LAN.

- The --progress flag gives continuous status information about the number of files copied, the network throughput, and the amount of time until the copy completes. We won't supply this argument in our automated rsync copies.

- The --partial flag tells rsync to keep partially downloaded files. When the copy of a large file is interrupted, rsync will continue upon the next invocation to pick up where it left off. If you're copying DVDs or other such large single files, you should use this every time in case the transfer is interrupted.

This rsync command uses SSH as the transport and requires us to log into the remote host. For our automated web updates, we use the rsync daemon to avoid exposing system accounts to potential attacks. We want as few trusts as possible that could potentially lead to an attacker gaining shell access to our servers—especially our cfengine master system.

When the rsync completes, immediately run the same rsync command again simply to see how long it takes this time. It will complete quickly because the two directories are already synchronized. In general, it will complete quickly when only a few files or only some contents in the files themselves change.

We also need to copy over our PHP build to the master system from where we built it on *rhmaster*:

```
# rsync -avze ssh --progress rhmaster:/usr/pkg/php-5.2.6/ \
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/php-5.2.6
```

Now move the `conf` directory out of `httpd-2.2.9/conf` and up one directory into the `pkg` directory, and rename it to `PROD/repl/root/usr/pkg/httpd-conf`. This will prevent the Apache configuration directory from being copied along with our binaries. We'll have continuity with the configuration files by separating the two. In addition, we'll have our rsync or cfengine copies ignore the `conf` directory so it can appear in the Apache directory tree without causing problems (but it's probably best to remove it, in order to avoid confusion later):

```
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/httpd-2.2.9
# mv conf ../httpd-conf/
```

In the file `PROD/repl/root/usr/pkg/httpd-conf/httpd.conf` we set these values:

```
ServerName shop.campin.net:80
DocumentRoot "/var/www/html"
# This should be changed to whatever you set DocumentRoot to.
#
<Directory "/var/www/html">
```

The comments were in the `httpd.conf` file already. We left them in this example to help you find the appropriate lines to modify. When you change the `DocumentRoot` setting, you need to modify this directory section to reflect the same name. It sets the proper access-control settings, default-directory index names, and some other settings that basically make your web contents visible to remote clients.

To automate the copy of the Apache binaries, copy the `cf.sync_web_content` task to a new one called `cf.sync_apache_binaries`:

```
# cp PROD/inputs/tasks/app/apache/cf.sync_web_content  \
PROD/inputs/tasks/app/apache/cf.sync_apache_binaries
```

Edit the task so that it has these contents:

```
directories:
        web_server::
                /usr/pkg mode=755 owner=root group=root inform=true


shellcommands:
    web_server.Hr00.Min00_05::
        "/usr/bin/rsync -aq --delete --exclude 'conf' \
          ${web_master}\:\:apache-2.2.9/ /usr/pkg/httpd-2.2.9"
                        timeout=600 inform=false umask=022


    "/usr/bin/rsync -aq --delete ${web_master}\:\:php-5.2.6/ /usr/pkg/php-5.2.6"
                        timeout=600 inform=false umask=022


  web_server::
    "/usr/bin/rsync -aq --delete ${web_master}\:\:httpd-conf/ /usr/pkg/httpd-conf"
                        timeout=600 inform=false umask=022


links:
        web_server::
                /usr/pkg/httpd          ->!     /usr/pkg/httpd-2.2.9
                /usr/pkg/httpd/conf     ->!     /usr/pkg/httpd-conf
```

We check for updated binaries only once per day, at midnight. We do check for updated Apache config files, however, every time cfagent is run. To put this task into action, add it to the PROD/inputs/hostgroups/cf.web_server hostgroup file.

Now we'll configure cfengine to automatically start up our newly distributed Apache server. Edit the task tasks/app/apache/cf.apache_start and modify it to look like this:

```
processes:
        web_server.redhat::
                "httpd"  restart "/usr/pkg/httpd/bin/apachectl start"
                            inform=true umask=022
```

■**Note** If you enabled SELinux on your Red Hat web-server systems, Apache will not work properly without SELinux modifications. We do not cover SELinux configuration in this book, but instead refer you to the Red Hat online documentation at https://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.2/html/Deployment_Guide/selg-overview.html.

Can you think of anything that might be lacking in this method of synchronizing the Apache, PHP binaries, and Apache configuration files? We hope you guessed it: when our Apache configuration files update, we don't automatically restart Apache. If we used cfengine for the copy, we could easily set a class that triggers a restart via a `shellcommands` or `processes` action.

So we have two choices:

- Process the output of the rsync command when called by cfengine, and look for strings that signal success. Exit codes from rsync aren't very meaningful; only output from the program can help you.

- Continue using the binary synchronization with rsync, but move the config-file copy into a cfengine `copy` section. This is probably the wiser option, and more robust because we don't have to hack together a script to parse the rsync output.

Now we have Apache up and running with automated configuration updates, but without automated restarts when the configuration files are updated. Next, we'll explore other data-sharing options without this problem.

Sharing Data with NFS

Network filesystems can be very convenient, but they can also give you headaches. When the server is up and running and the network is operating properly, network filesystems are good for many situations. You can access the same applications and data from multiple systems. You can modify the files on any system, and the changes become instantly available on all other systems.

Problems with network filesystems begin to appear when you have server or network problems. Network filesystems do not operate at 100 percent efficiency when these problems occur. In many cases, the network filesystem might not work at all and could even hang the UNIX/Linux host.

We previously put NFS into use in our example environment in Chapter 7, although without much discussion of which situations tend to be a good fit for NFS. In this chapter, we discuss the pros and cons of using NFS for storing program data as well as program executables.

The NFS is one of the oldest network filesystems still in common use today. Its single greatest advantage is its native support in most, if not all, UNIX variants. It is also generally easier to configure and use than other network filesystems. As with just about any network filesystem, you can make changes on the server or on any client and have them be immediately available on all other systems.

The data on an NFS filesystem is physically located on only one server. Every client mounts the filesystem directly from that server. If you have a significant number of clients or if they're highly active, you might run into performance problems. You will either have

to upgrade the server or move a portion of the data into separate NFS shares on separate servers to alleviate these problems. All NFS clients have some amount of caching capabilities to help increase performance and reduce the load on the server. Some operating systems, however, will have better caching support than others.

■**Tip** We provide a brief introduction to NFS in this book. For more information, check out *Managing NFS and NIS, Second Edition* by Hal Stern, Mike Eisler, and Ricardo Labiaga (O'Reilly Media Inc., 2001).

The biggest disadvantage of NFS is that it relies on the network and a single server. Depending on the client's implementation and mount options used, any network or server downtime can cause the client system to hang, particularly when a process is using the network filesystem. If critical data is shared via NFS, a problem with the server might make all client systems inoperable.

With NFS, a user should have the same user ID on all systems. All file ownership on an NFS filesystem—well, any filesystem, really—is assigned by user IDs (UIDs) and group IDs (GIDs). If a user has a different UID on each system, then he or she will not be able to modify his or her own files, and possibly will able to modify somebody else's. This is one of the reasons that we unified the UIDs in our example environment early on.

NFS uses Remote Procedure Calls (RPCs), so the server and all clients need to run the portmap daemon at all times. The portmap daemon allows RPC connections to be initiated between systems. The use of RPCs makes it difficult to use NFS through most firewalls. NFS also uses a variety of helper programs, such as `mountd` and a locking daemon.

For these reasons, you might find it difficult to configure NFS and use it in your environment. The Internet Engineering Task Force's version 4 of the NFS protocol (NFSv4) addresses most of the protocol's deficiencies (see `http://www.nfsv4.org`). For example, it specifies mandatory access-control lists and other security settings such as encryption, it's stateful, and it operates over a single TCP port. Our experience with it in production environments tells us that it's still a good idea to stick with a single code base for client and server systems; for example, we recommend using Solaris systems only or another single OS-vendor configuration. In mixed environments, we have had to force all systems back to NFS version 3. This doesn't mean you can't make NFSv4 work with some careful testing and research on Internet mailing lists and newsgroups, but the effort is not trivial.

Configuring the NFS Server

The NFS server usually needs to run one or more daemons. Exactly what daemons need to run depends on the operating system, so you need to consult your OS's documentation for details.

Any server, however, needs to specify what portion of its local filesystem needs to be shared, or exported, to NFS clients. You accomplish this on Linux systems with the `/etc/exports` file.

Here is a simple example:

```
/export/home foobar.example.com(rw,root_squash)
/mnt/cdrom   192.168.0.0/255.255.0.0(ro,no_root_squash)
```

For this example, we assume that some home directories reside under the `/export/home/` directory. So, to allow systems to mount users' home directories remotely, we export the `/export/home/` directory (and everything under it) to the host *foobar.example.com*. The CD-ROM on this system is also exported to all systems that have an IP address beginning with `192.168`.

Each line ends with one or more options in parentheses. The available options are different on every operating system, but here are a few basic ones you can expect to find almost everywhere:

`ro`: Clients can only read from this filesystem, which is the default.

`rw`: Clients can read and write to this filesystem.

`root_squash`: The `root` user on the client system does not have `root` privileges on the network filesystem. So, if a file is owned by `root`, a client cannot delete it. If a file is not publicly readable, it cannot be read by `root` on the client system. This is the default.

`no_root_squash`: The `root` users on clients do have special access on the filesystem.

The `root_squash` option is very important for system security. If a user has `root` access to a client system, but not the server, he or she can still bypass system security. A user could simply copy the bash shell, for example, onto the network filesystem and make it setuid `root` (which means anybody who runs the program runs it as `root`). The user can do this because the copy is taking place on the client and the user has `root` access on that system. This user can now log in to the NFS server with a regular user account, execute that Bash shell, and gain full `root` access to the server. So, you should use the `root_squash` option whenever possible, especially if not-so-trusted users have `root` access on any systems that are allowed to mount your NFS resources.

Configuring the NFS Client

Once portmap is running, any client with appropriate permission can mount an NFS partition from a server with the following command:

```
# mount server.mydomain.com:/export/home/kirk /home/kirk
```

On some systems, you might need to specify a `-t nfs` or `-F nfs` switch to the `mount` command. See the `mount` man page on your system for more details.

A client can also automatically mount NFS filesystems at startup by placing entries in the standard `/etc/fstab` filesystem table (`/etc/vfstab` on Solaris).

Sharing Program Binaries with NFS

We can share over NFS the Apache binaries that we built for Red Hat, or any other compiled programs that we might need. Advantages of this approach include less need for local storage on application systems and rapid distribution of software updates. As soon as the NFS server is updated, all the clients are updated as well.

These reasons aren't very compelling in modern environments. Most SAs remember when a large hard disk had less (sometimes far, far less) than 10GB of storage, while a relatively inexpensive new system today will have at least 100GB of local storage or perhaps a half terabyte. You should store frequently used program binaries locally, as disk space is cheap and a network interruption shouldn't block access to important binaries.

You might need to keep application data in sync across many systems, so NFS is a good fit in such a scenario. In addition, you might want to place infrequently used applications on an NFS share so that users can run it only on occasion, and thereby save disk space on other systems.

Server Setup

We've already set up NFS exports in our example network for NFS-mounted home directories. Let's set up a system of application shares meant for different hardware and OS platforms to utilize a collection of binary applications. We'll utilize the same NFS server as before, the host named *aurora* (named after the Spacer planet in Isaac Asimov's Robot Series; see `http://en.wikipedia.org/wiki/Aurora_(planet)`).

We'll want to automate the NFS export of `/export/pkg`. Add this line to `PROD/inputs/classes/cf.main_classes` so that we have a new server role:

```
binary_server          = (     aurora )
```

Next, create a new task at the location `PROD/inputs/tasks/app/nfs/cf.export_pkg_share` with these contents:

```
control:
        any::

                addinstallable = ( restart_nfs )
```

```
directories:
        binary_server.(solaris|solarisx86)::
                /export/pkg/sunos_sun4u mode=755 owner=root group=root inform=false
                /export/pkg/i686.redhat  mode=755 owner=root group=root inform=false
                /export/pkg/i686.debian  mode=755 owner=root group=root inform=false

shellcommands:
        binary_server.restart_nfs.(solaris|solarisx86)::
                "/usr/sbin/svcadm restart network/nfs/server"
                        timeout=60 inform=true

editfiles:
        binary_server.(solaris|solarisx86)::
                { /etc/dfs/dfstab
                        AppendIfNoSuchLine  "share -F nfs -o rw,anon=0 /export/pkg"
                        DefineClasses           "restart_nfs"
                }
```

On our NFS server *aurora*, we created a collection of directories in /export/pkg meant for each of our three platforms:

- sunos_sun4u: Solaris hosts running on SPARC hardware

- i686.redhat: Red Hat hosts running on 32-bit, x86-based hardware

- i686.debian: Debian hosts running on 32-bit, x86-based hardware

Create a file called PROD/inputs/hostgroups/cf.binary_server with these contents:

```
import:
        any::
                tasks/app/nfs/cf.export_pkg_share
```

Then set up the import for the binary_server role in the file PROD/inputs/hostgroups/cf.hostgroup_mappings by adding this line:

```
binary_server::                         hostgroups/cf.binary_server
```

The next time cfexecd runs cfagent (always within the next 20 minutes, at our example site), the NFS share will be configured on *aurora*.

Client Setup

Once again, we'll utilize the automounter daemon on our clients to handle mounting NFS shares. Using the automounter gives us much more flexibility than any scheme we invent using static mounts, whether using cfengine's direct NFS-mounting abilities or cfengine edits to /etc/fstab or /etc/vfstab files.

■**Caution** You'll see the code-continuation character (➥) within this section's code. This character signifies that the line in which it appears is actually a single line, but could not be represented as such because of print-publishing restrictions. It is important that you incorporate the line in question as a single line in your environment. You can download all the code for this book from the Downloads section of the Apress web site at http://www.apress.com.

We'll modify our master automounter files (auto.master on Linux and auto_master on Solaris) to import a new map file, and use cfengine to automatically create the file with the appropriate contents for that type of system. First, add this line to the master automounter files on the cfengine master:

```
/mnt/pkg          /etc/auto_mnt_pkg
```

We use the same map file name on both Linux and Solaris, to make the cfengine config files slightly shorter. Next, create a task that uses editfiles to create the map file, a file at PROD/inputs/tasks/os/cf.create_autofs_mnt_pkg with these contents:

```
control:

        old_binary_server       = ( made_up_name )
        binary_server           = ( aurora )

editfiles:

        i686.debian::
                { /etc/auto_mnt_pkg
                        AutoCreate

                        AppendIfNoSuchLine              "*        -nolock,rsize=➥
32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5    $(binary_➥
server):/export/pkg/i686.debian/&"
```

```
                            DeleteLinesContaining    "*       -nolock,rsize=32767,➡
wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5     $(old_binary_➡
server):/export/pkg/i686.debian/&"


                            DefineClasses "restartautofs"
                }

        i686.redhat::
                { /etc/auto_mnt_pkg
                        AutoCreate
                        AppendIfNoSuchLine                "*       -nolock,rsize=➡
32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5     $(binary_➡
server):/export/pkg/i686.redhat/&"


                        DeleteLinesContaining              "*       -nolock,rsize=➡
32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5     $(old_binary_➡
server):/export/pkg/i686.redhat/&"
                        DefineClasses "restartautofs"
                }

        sunos_sun4u::
                { /etc/auto_mnt_pkg
                        AutoCreate
                        AppendIfNoSuchLine                "*       -nolock,rsize=➡
32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5     $(binary_➡
server):/export/pkg/sunos_sun4u/&"


                        DeleteLinesContaining              "*       -nolock,➡
rsize=32767,wsize=32767,proto=tcp,hard,intr,timeo=8,nosuid,retrans=5     $(old_➡
binary_server):/export/pkg/sunos_sun4u/&"
                        DefineClasses "restartautofs"
                }

directories:
        any::
                /mnt/pkg mode=755 owner=root group=root inform=false
```

Remember that the restartautofs class triggers an automounter restart in the task
PROD/inputs/tasks/os/cf.sync_autofs_maps, so we don't need to re-create the restart logic
in this task. We can simply define the class in this task, and the restart happens via the
other task (cf.sync_autofs_maps).

We set up support for moving the `binary_server` duties to another host, even though we have no immediate plans to perform such a move. It makes sense to be ready in case the host *aurora* fails and can't be replaced. We could use a DNS alias to perform this with much less complexity, but we wanted to explore `editfiles` usage more fully in this section. The `editfiles` actions in this task handle the deletion of old entries that are identical to the entries to mount our current NFS server *aurora*, but only if the entry contains the string contained in the variable `old_binary_server`.

To activate this task, add it to the `PROD/inputs/hostgroups/cf.any hostgroup` file.

The choice of what to put on the binary server is entirely site-dependent. It's a good idea to copy the files there using cfengine (or perhaps Subversion, as demonstrated later in the chapter), so that if you have to rebuild the binary server, cfengine can pull the needed files back again through a cfengine copy or a Subversion checkout. And you might want to use rsync's mirroring capabilities to keep the binary server updated. Again, the choice is yours, but we wanted to inform you of the pros and cons of different data-sharing methods so that you can make informed decisions on your own.

Once you place files and directories into the binary server's shared directories, you can utilize them as though they are local files, to be mounted on demand by the automounter. We created the directory `/export/pkg/i686.debian/bind` on *aurora*, and placed the programs `named-checkconf` and `named-checkzone` there. We copied them from `/usr/sbin` on the host *etchlamp*, as they are part of the `bind9` package. You'll benefit from utilizing these programs from other hosts, especially the cfengine master system. This way, when we add entries to the BIND zone files, we can syntax-check them:

```
# cd /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/bind/debian-ext/
# /mnt/pkg/bind/named-checkzone campin.net db.campin.net
zone campin.net/IN: loaded serial 2008080601
OK
```

The `named-checkzone` is usually used with only two arguments: the name of the DNS zone, and the name of the zone file.

This syntax check lets us know that the zone file we edited won't be rejected by BIND when it's distributed by cfengine and loaded by the nameserver host. We recommend that you run this check every time you make zone-file edits. The program `named-checkconf` performs the same role, but for BIND configuration files.

If you place Apache and PHP in this directory, you'll need to modify the startup and restart definitions in your task files appropriately. We won't demonstrate this because we don't think running such important programs from an NFS mount is the best idea. You should copy the programs to the system's local drive, for one simple reason: an SA shouldn't have to get up at night to respond to an NFS-related problem with the production web server. As we said before, disk space is cheap, so take advantage of it. NFS mounts are usually a better fit for user home directories and utility programs.

Sharing Data with cfengine

We intend to copy the Apache and PHP programs to our Red Hat web server using cfengine. We've used cfengine quite a bit at this point, so we don't think you'll see any surprises in this chapter regarding its general use. Let's use the same file layout from the rsync-based copy of Apache that we configured in the last section. Simply change these lines in PROD/inputs/tasks/app/apache/cf.sync_apache_binaries:

```
shellcommands:
      web_server.MinOO_05::
            "/usr/bin/rsync -aq --delete --exclude 'conf' \
              ${web_master}\:\:apache-2.2.9/ /usr/pkg/httpd-2.2.9"
                    timeout=600 inform=false umask=022

            "/usr/bin/rsync -aq --delete ${web_master}\:\:php-5.2.6/ \
                    /usr/pkg/php-5.2.6"
                    timeout=600 inform=false umask=022

            "/usr/bin/rsync -aq --delete ${web_master}\:\:httpd-conf/ \
                  /usr/pkg/httpd-conf"
                    timeout=600 inform=false umask=022
```

to these:

```
control:
      any::
            addinstallable = ( create_homedirs enable_nfs )

copy:
      web_server.MinOO_05::
            $(master)/repl/root/usr/pkg/httpd-conf
                              dest=/usr/pkg/httpd-conf
                              mode=644
                              r=inf
                              purge=true
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
                              define=reload_httpd
```

```
        $(master)/repl/root/usr/pkg/httpd-2.2.9
                        dest=/usr/pkg/httpd-2.2.9
                        mode=755
                        r=inf
                        purge=true
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        define=restart_httpd


        $(master)/repl/root/usr/pkg/php-5.2.6
                        dest=/usr/pkg/php-5.2.6
                        mode=755
                        r=inf
                        purge=true
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        define=restart_httpd

shellcommands:
    redhat.web_server.restart_httpd::
            "/usr/pkg/httpd/bin/apachectl restart"
                    timeout=60 inform=true


    redhat.web_server.reload_httpd::
            "/usr/pkg/httpd/bin/apachectl reload"
                    timeout=60 inform=true
```

We trigger a class that reloads Apache when the configuration files are updated. In this setup, we set a class that fully restarts Apache when the binaries for Apache or PHP are updated, because we don't want to copy new programs out and not start using them right away.

Note that we set the synchronization of the Apache and PHP directories to occur hourly. In all likelihood, you'll upgrade Apache only a few times per year. Having cfengine crawl all 1,600 files in the PHP and Apache master directories as well as the destination

directories every hour seems wasteful. We recommend adding an hour class, so that it reads like this:

```
web_server.Hr00.Min00_05::
```

The new setting will make the synchronization run once per day at midnight. You should realize that the performance of your cfengine runs will matter quite a bit in the long run. Right now, a `cfagent` run completes in a short amount of time. If we keep adding unnecessary file copies across large directories, we'll slow down the cfengine runs needlessly, and steal CPU time away from processes that might really need it.

Sharing Data with Subversion

The first edition of this book utilized CVS for revision control–based file distribution. In this edition, we use Subversion, a revision-control system written to replace CVS. It is not the only option out there, but it's a solid choice for several reasons:

- System accounts are not required for access to the Subversion repository, when paired with Apache.

- It has strong access-control options when paired with Apache.

- CVS tracks only the versioning of files, while Subversion tracks and fully version-controls everything in the repository—both files and directories (and even symlinks on UNIX).

- Subversion supports atomic commits: a collection of files is either fully checked in as a group, or not committed at all.

- Subversion deals well with binary files. CVS can do it, but not as gracefully.

The authoritative reference book on Subversion is available for free on the web: `http://svnbook.red-bean.com/`.

Automating Deployment of Your Subversion Server

We'll set up our main infrastructure host, *etchlamp.campin.net*, as our network's Subversion server. You'll recall that it is running Debian GNU/Linux. To get Subversion installed at initial-system installation time, add these packages to the `WEB` class package list in FAI (`/srv/fai/config/package_config/WEB`):

```
subversion
subversion-tools
libsvn-notify-perl
db4.4-util
patch
libapache2-svn
ssl-cert
mailx
```

It is up to you whether to manually install the preceding packages, or reimage the host. You'll need them installed before moving on. On the cfengine master, create the directory `PROD/repl/root/etc/apache2`. All the files that we need will be copied from this directory. Before we do anything else, we need to add this line to `PROD/repl/root/etc/bind/debian-ext/db.campin.net` so that we can refer to our Subversion server as "svn.campin.net":

```
svn     300     IN      CNAME   etchlamp
```

Then run `named-checkzone` against the zone file to check for errors:

```
# /mnt/pkg/bind/named-checkzone campin.net \
/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/bind/debian-ext/db.campin.net
zone campin.net/IN: loaded serial 2008080800
OK
```

Then we just have to wait up to 20 minutes for the new DNS zone to go live. We're not in a hurry, because we're doing this a little while before we will access our new Subversion server. We'll work on other things while waiting.

We need to set up the Secure Sockets Layer (SSL) certificate for Apache ahead of time with an interactive command. Run `/usr/sbin/make-ssl-cert` to generate the key, and we'll copy the certificate using cfengine later—after generating it with this command:

```
# mkdir /etc/apache2/ssl
# /usr/sbin/make-ssl-cert /usr/share/ssl-cert/ssleay.cnf /etc/apache2/ssl/apache.pem
# scp -r /etc/apache2/ssl \
goldmaster: /var/lib/cfengine2/masterfiles/PROD/repl/root/etc/apache2/
```

This will put the two SSL key files generated in *etchlamp's* `/etc/apache2/ssl` directory into a new directory on the cfengine master system: `/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/apache2/ssl`.

Next, we'll run the htpasswd command on *etchlamp* to set up our first user account, and copy the file to the cfengine master as well (to the directory PROD/repl/root/etc/apache2/):

```
#  htpasswd -c /etc/apache2/dav_svn.passwd nate
```

Choose a password when prompted. You want to use the -c argument only the very first time you use a file controlled by htpasswd. It initializes a new file, and will overwrite an existing one. Be careful.

Next, we'll create a file at the location PROD/repl/root/etc/apache2/sites-available/svn.campin.net. This is an Apache configuration file with these contents:

```
NameVirtualHost *:443

<VirtualHost *:443>
        ServerName      svn.campin.net
        ServerAlias     svn

        DocumentRoot    /var/www
        ServerAdmin     admins@foo.bar
        ErrorLog             /var/log/apache2/svn-error.log
        CustomLog       /var/log/apache2/svn-access.log combined

        <Location /svn>
                DAV svn
                SVNParentPath /var/svn/repository
                AuthType Basic
                AuthName "Campin.net Subversion Repository"
                AuthUserFile /etc/apache2/dav_svn.passwd
                AuthzSVNAccessFile /etc/apache2/svn_accessfile
                Require valid-user

                <LimitExcept GET PROPFIND OPTIONS REPORT>
                 Satisfy all
                </LimitExcept>
        </Location>

        SSLEngine on
        SSLCertificateFile /etc/apache2/ssl/apache.pem

</VirtualHost>
```

We also place a modified `ports.conf` file in `PROD/repl/root/etc/apache2`, with this single line of content:

```
Listen 443
```

Our Subversion Apache server will support HTTPS only. The final file to create is `PROD/repl/root/etc/apache2/svn_accessfile`, with these contents:

```
[groups]

admins = nate,kirk

[cfengine:/]
@admins = rw

[binary-server:/]
@admins = rw
```

This file is used to grant fine-grained access to the repositories on the server. For now, we grant full access to the nate and kirk accounts. The portion in square brackets where we specify only the / character can specify paths in each repository, and set precise controls on the sort of access allowed. This might come in handy later.

Next, create `PROD/inputs/tasks/app/svn/cf.setup_svn_plus_apache` with these contents:

```
classes:  # synonym groups:
        have_ssl_load   = ( "/usr/bin/test -f /etc/apache2/mods-enabled/ssl.load " )
        have_binary_repo = ( "/usr/bin/test -d /var/svn/repository/binary-server " )
        have_cfengine_repo   = ( "/usr/bin/test -d /var/svn/repository/cfengine " )

control:
        any::
                addinstallable          = (     restart_apache2  )

copy:
        svn_server.debian::
                $(master_etc)/apache2
                                dest=/etc/apache2
                                mode=444
                                r=inf
                                purge=false
                                owner=root
```

```
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_apache2


directories:
        svn_server.debian::
            /var/svn  mode=770 owner=www-data group=www-data inform=true
            /var/svn/repository  mode=770 owner=www-data group=www-data inform=true


processes:
        svn_server.debian::
                # start it when it's not running
                "/usr/sbin/apache2" restart "/etc/init.d/apache2 start"
                    inform=true umask=022


shellcommands:
        svn_server.debian.!have_cfengine_repo::
                "/bin/su www-data -c \"/usr/bin/svnadmin create \
                 /var/svn/repository/cfengine\" "
                        #owner=33 # www-data user's UID
                        timeout=60
                        umask=022


        svn_server.debian.!have_binary_repo::
                "/bin/su www-data -c \"/usr/bin/svnadmin create \
                   /var/svn/repository/binary-server\" "
                        #owner=33 # www-data user's UID
                        timeout=60
                        umask=022


        svn_server.debian.restart_apache2::
                "/etc/init.d/apache2 restart"
                        timeout=60
                        umask=022


        svn_server.debian.!have_ssl_load::
                "/usr/sbin/a2enmod ssl"
                        timeout=60
                        umask=022
```

```
disable:
        svn_server.debian::
                /etc/apache2/sites-enabled/000-default


links:
        svn_server.debian::
                /etc/apache2/sites-enabled/svn.campin.net ->!
                    /etc/apache2/sites-available/svn.campin.net
```

This task is fairly straightforward. We have a directory structure on the cfengine master that's copied out recursively, overwriting any files with the same names that exist prior to the copy. We have already shown the contents of all the files that we copy out. We disable the default Apache virtualhost added by the Debian apache2 package, and use only our *svn.campin.net* configuration.

We take additional steps in the task to enable the mod_ssl Apache module, and to create the two Subversion repositories that we want right away. We don't do anything with the "cfengine" repository now, but we'll make use of it in later chapters.

It was necessary to use the su command to create the missing repositories because when we used cfengine's feature to set the user that commands run as (see commented entries), the svnadmin utility attempted—and failed—to source configuration files in /root, the root user's home directory. Executing the svnadmin create command under the su command fixed this error.

Finally, we link the *svn.campin.net* virtualhost into the sites-enabled directory, which activates the configuration in Apache.

We need to define the svn_server class in the file PROD/inputs/classes/cf.main_classes with this line:

```
svn_server              = (    etchlamp )
```

Create PROD/inputs/hostgroups/cf.svn_server and put this in it:

```
import:
        any::
                tasks/app/svn/cf.setup_svn_plus_apache
```

Finally, add the hostgroup import to the PROD/inputs/hostgroups/cf.hostgroup_ mappings file by adding this line:

```
svn_server::                    hostgroups/cf.svn_server
```

Now wait for cfexecd to configure Subversion and Apache on *etchlamp* at the next scheduled run, or log in and run cfagent yourself to hurry things up.

Now that cfengine has set up the repositories and Apache for us, you can use Subversion over HTTPS from anywhere on our network.

Using Subversion

The basic usage of Subversion is easy to learn. For our initial walkthrough, we'll cover only basic tasks—adding files and file trees, and working with those files once they're under Subversion control. In later chapters, we'll cover more advanced topics such as branching and merging.

To place the Subversion client on all future Solaris installs, add the string subversion to the list of packages installed by pkg-get in the "run-once" script that's put in place by the JumpStart postinstall script.

For the host *aurora* we can simply run:

```
# pkg-get install subversion
```

Importing files is the first task to perform with a new repository. We import the binary-server tree from the NFS server *aurora* using this procedure:

```
# svn --username=nate import /export/pkg https://svn.campin.net/svn/binary-server \
-m"initial import"
Error validating server certificate for 'https://svn.campin.net:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: svn.campin.net
 - Valid: from Fri, 08 Aug 2008 09:22:06 GMT until Sun, 07 Sep 2008 09:22:06 GMT
 - Issuer: Ops, campin.net, SF, California, US
 - Fingerprint: 97:65:8b:7b:88:cb:a0:ba:46:d2:1a:27:44:97:93:ed:db:c6:12:f4
(R)eject, accept (t)emporarily or accept (p)ermanently? p
Authentication realm: <https://svn.campin.net:443> Campin.net Subversion Repository
Password for 'nate':
Adding         /export/pkg/i686.redhat
Adding         /export/pkg/sunos_sun4u
Adding         /export/pkg/i686.debian
Adding         /export/pkg/i686.debian/bind
Adding  (bin) /export/pkg/i686.debian/bind/named-checkconf
Adding  (bin) /export/pkg/i686.debian/bind/named-checkzone

Committed revision 1.
```

You use the `svn import` command when you have a group of files that you want to begin tracking in Subversion. After the import is finished, the original files still aren't under Subversion control. We need to check out the repository in order to start working with the files in the repository, as shown here:

```
# cd /export
# mv pkg pkg.bak
# svn --username=nate co https://svn.campin.net/svn/binary-server pkg
A    pkg/i686.redhat
A    pkg/sunos_sun4u
A    pkg/i686.debian
A    pkg/i686.debian/bind
A    pkg/i686.debian/bind/named-checkconf
A    pkg/i686.debian/bind/named-checkzone
Checked out revision 1.
# svcadm restart nfs/server
```

We move the original `pkg` directory out of the way and check out the "binary-server" repository—in its entirety; we have a dead-simple repository layout for it—to a directory named `pkg`. Afterward, we need to restart the NFS server so that it serves out the re-created directory. Now when we add or change any files or directories in this tree, we'll want to let Subversion know about it.

After we create the file `foo`, the `svn status` command shows us that we have files not found in the repository. It signals this condition by showing the ? symbol to the left of the file name:

```
# cd /export/pkg
# ls
./          ../          .svn/          i686.debian/  i686.redhat/  sunos_sun4u/
# touch foo
# svn status
?      foo
```

We can add and commit the new file to the repository with these commands:

```
# svn add foo
# svn commit foo
```

We aren't required to add files to Subversion that we add into this directory tree, but if we don't, the file will be lost if we set up a new server. From this point on, we can distribute this directory tree using Subversion inside cfengine `shellcommands`, probably as the final step in setting up the `binary_server` NFS server role in cfengine. If we end up doing

automated checkouts, a read-only user should be set up for automated access to the repository (for security reasons).

Revision control of these binaries will be useful in case any upgrades to the files cause problems later. As long as each version is checked into Subversion, we can easily roll back the change.

The *svn.campin.net* Subversion configuration is fully automated, although it will of course lack any actual repository data when initially configured.

If you completely lose your Subversion server (e.g., due to hardware failure) and you don't have any backups, you can reimage it and you'll get the bare repositories back again due to cfengine's configuration. At that point, you could import a previously checked-out copy (from before the crash), and Subversion will ignore all the .svn directories when the import is done. It will simply create a new "revision 1" based on what you import, just like our initial import of the tree.

This somewhat workable disaster-recovery scenario isn't a substitute for proper back-ups because you'll lose all your revision history. At least you can get going again without having to configure your Subversion server manually. We'll briefly address backups in a later chapter, although not nearly with the treatment they deserve. We recommend *UNIX Backup and Recovery* by W. Curtis Preston (O'Reilly Media Inc., 1999).

We'll finish up this chapter by showing a way to get the Subversion client installed on all your Red Hat hosts. We won't install it when the host is initially imaged; instead, we'll have cfengine use yum to install it. We have used the cfengine packages action to determine if our site's DNS server has the Debian bind9 package installed, but we haven't used it to actually install any packages.

Cfengine requires that an installation command be defined for each platform on which the packages action is used. For Red Hat, we'll use this entry:

```
DefaultPkgMgr = ( rpm )
RPMInstallCommand = ( "/usr/bin/yum -y -q install %s" )
```

This command will utilize the yum package-installation tool to install packages from cfengine. The yum tool can use multiple network repositories to find packages, and will automatically resolve package dependencies and install packages to satisfy those dependencies. We'll add these entries to the file PROD/inputs/control/cf.control_cfagent_conf. We already have a Red Hat section that looks like this:

```
        redhat::
                cf_base_path    = ( /usr/pkg/cfengine/sbin )
                workdir         = ( /var/cfengine )
                client_cfinput  = ( /var/cfengine/inputs )


        !debian.!redhat.!(solaris|solarisx86)::
```

We'll add our entries into the `redhat` section, so that it looks like this:

```
redhat::
        cf_base_path   = ( /usr/pkg/cfengine/sbin )
        workdir        = ( /var/cfengine )
        client_cfinput = ( /var/cfengine/inputs )

        DefaultPkgMgr = ( rpm )
        RPMInstallCommand = ( "/usr/bin/yum -y -q install %s" )


!debian.!redhat.!(solaris|solarisx86)::
```

We will create a task to install the Subversion client. The task file is named `PROD/inputs/tasks/os/cf.install_svn`, with these contents:

```
packages:
      redhat::
          subversion
              action=install
```

We'll add the task to the `cf.any` hostgroup, so that we can easily support Subversion installation for other platforms that we'll add to our environment later. The next time `cfagent` runs on our Red Hat hosts, the Subversion RPM will be installed if it's not present on the system. If you run `cfagent -qv` on a Red Hat system without the Subversion RPM, you'll see output like this (along with a lot of other output):

```
RPM Package subversion not installed.
Package manager will be invoked as /usr/bin/yum -y -q install
BuildCommandLine(): Adding package 'subversion' to the list.
cfengine:rhlamp: Installing package(s) using '/usr/bin/yum -y -q install subversion'
cfengine:rhlamp: Packages installed: /usr/bin/yum -y -q install subversion
```

NFS and rsync and cfengine, Oh My!

In this chapter, we've covered many different ways to share and copy data. Along the way, we managed to configure a web site at our example site. We've made it fairly clear which way we like to copy program binaries to clients (cfengine), but our intention is never to make your choices for you.

We want you to know all the common ways to move data around on a network, and to make the choice that best suits your needs. Even sites running similar operating systems and applications might have very different requirements around performance and

security, as well as differences in how much time they're willing to spend investigating new file-distribution methods.

System administrators end up becoming very adept at copying files from many different sources to many different destinations. If you're at or near the beginning of your SA career, study the options and tools in this chapter on your own and learn them well. The time will be well spent.

At this stage in the construction of our example infrastructure, it might seem like we're almost finished. After all, we had the required infrastructure to deploy our first application, right? Well, yes, we have a functional infrastructure, but it's still in its early stages. We're lacking in several key areas, and we'll address some of our basic reporting needs in the next chapter.

◼ ◼ ◼

Generating Reports and Analyzing Logs

You need to know about errors on the systems in your environment before they turn into major problems. You also need the ability to see the actions your automation system is performing. This means you'll need two types of reporting:

- Log reports that capture bad strings, report them immediately, and ignore the rest

- Log reports that ignore "okay" strings and report on what's left

You want to know right away if a system has serious hardware issues or major application issues. We're going to run a reporting system that looks for unwanted words or phrases. Here is one such unwanted message:

```
using specific query-source port suppresses port randomization and can be insecure
```

This particular message means that you're running a version of BIND that works around a serious vulnerability, but that a configuration directive overrides the workaround. If this were going on in your network, you'd want to know as soon as possible. We'll use real-time alerting to pick up on this condition as well as others.

Reporting on cfengine Status

You have two main ways of tracking the actions and changes that cfengine makes across your infrastructure. First, you set `syslog=on` globally at your site, so that cfengine logs all actions to syslog. Second, you have the output of `cfagent` itself, which is (of course) not included in the syslog entries. When `cfexecd` runs `cfagent` (as it does at our site), it always stores the output, including the output of any commands run by `cfagent`, in the `$workdir/outputs` directory. Also, `cfexecd` e-mails the output of `cfagent` to the `sysadm` e-mail address as defined in `cfagent.conf` (or as in our case, a file imported from `cfagent.conf`).

We are very interested in the contents of `$workdir/outputs`, and would like to aggregate them centrally. We can later run interactive checks such as simple `grep` searches, or directly view the files when we need to do some investigation. For monitoring purposes, we can write scripts to flag and e-mail particular output-file contents to the administrators. This sort of scheme is useful if you'd rather use custom reporting instead of the e-mail functionality of `cfexecd`.

The first step is to get the `outputs` directory contents from all hosts aggregated to a single host. This presents something of a challenge, because cfengine uses a pull model. You don't want to keep an explicit list of all your systems and have one system try to pull the `outputs` directory contents from each. You'd rather have each system be responsible for pushing its `outputs` directory contents to a central location. We can take advantage of the rsync daemon that we placed on our cfengine master to accomplish this.

This approach brings with it some important security considerations:

- We need to grant access via a mechanism that won't allow a malicious user to access or destroy parts of the filesystem on the cfengine master.

- We need to prevent hosts from overwriting one another's files, whether accidentally or maliciously.

We can `chroot` the incoming copy with a feature of rsync, but that won't work in our case because we run our rsync daemon as a non-`root` user. Furthermore, we *don't* want to start running it as root because software bugs such as buffer overflows would result in remote attackers' ability to execute code as root on our system. We'd rather run the daemon as a non-`root` user and protect ourselves another way. Rsync allows a `pre-exec` script to be run before the copy is initiated, and we'll use that functionality to do some basic security checks.

■**Caution** You'll see the code-continuation character (➥) in some of this chapter's code sections. This character signifies that the line in which it appears is actually a single line, but could not be represented as such because of print-publishing restrictions. It is important that you incorporate the line in question as a single line in your environment. You can download all the code for this book from the Downloads section of the Apress web site at `http://www.apress.com`.

First, we add this section to the *goldmaster* `rsyncd.conf`, located in our `masterfiles` repository at `PROD/repl/root/etc/rsync/rsyncd.conf-www`:

```
[outputs-upload]
        comment = cfexecd outputs dir uploads
        path = /var/log/cfoutputs
        use chroot = no
        max connections = 400
        lock file = /var/tmp/rsyncd5.lock
        read only = no
        write only = yes
        list = no
        hosts allow = *.campin.net
        hosts deny = 0.0.0.0/0
        timeout = 3600
        refuse options = delete delete-excluded delete-after ignore-errors \
                                   max-delete partial force
        dont compress = *
        incoming chmod = Do-rwx,Fo-rwx
        pre-xfer exec = /usr/local/bin/rsync-outputs-dir-pre-exec
```

We define the `pre-xfer exec` script location, allow incoming copies via the `write only`
setting, and set restrictive permissions on the copied files via the `incoming chmod` setting.

■**Note** The `refuse options` option allows you to specify a space-separated list of rsync command-line
options that will be refused by your rsync daemon. We utilize it to keep clients from deleting files or from
leaving partially copied files.

On *goldmaster* we create the directory `PROD/repl/root/usr/local/bin`, and put a script
into it named `rsync-outputs-dir-pre-exec`, with these contents:

```
#!/bin/sh
PATH=/bin:/usr/bin

if [ -z "$RSYNC_REQUEST" -o -z "$RSYNC_HOST_NAME" ]
then
        echo "We need to run in a rsyncd pre-exec environment."
        exit 1
fi

HOST_NAME=`basename "$RSYNC_HOST_NAME" .campin.net`
```

```
case $RSYNC_REQUEST in
outputs-upload/${HOST_NAME}|outputs-upload/${HOST_NAME}.campin.net)
        # We need to rsync to a path with our hostname in it only, looks good
        :
        ;;
*)
        echo "You can only upload to a path based on your own hostname."
        exit 1
        ;;
esac
```

This script performs a simple check (using a case statement) to make sure the path that the client is copying to matches what we expect. We allow a client to copy only to a directory matching either its short or fully qualified name. Note that this scheme relies on the security and integrity of the DNS. You could easily modify this technique to use IP addresses instead. Feel free to implement it that way at your site, especially if you don't control your DNS servers.

Next, we'll enhance our current rsync server task at PROD/inputs/tasks/app/rsync/cf.enable_rsync_daemon so it looks like this:

```
classes:  # synonym groups:
        have_usr_bin_rsync       = ( "/usr/bin/test -f /usr/bin/rsync " )
        have_etc_inetd_conf      = ( "/usr/bin/test -f /etc/inetd.conf " )

control:
        any::
                AllowRedefinitionOf     = ( rsyncd_conf )
                AddInstallable               = ( hup_inetd  )

        web_master::
                rsyncd_conf     = ( "rsyncd.conf-www" )

copy:
        web_master::
                $(master_etc)/rsync/rsyncd.conf-www
                                dest=/etc/rsyncd.conf
                                mode=444
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
```

```
                  $(master)/repl/root/usr/local/bin/rsync-outputs-dir-pre-exec
                              dest=/usr/local/bin/rsync-outputs-dir-pre-exec
                              mode=555
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
editfiles:
      web_master.have_etc_inetd_conf::
              { /etc/inetd.conf
                      AppendIfNoSuchLine      "rsync   stream  tcp     nowait  ➥
daemon   /usr/bin/rsync rsyncd --daemon --config=/etc/rsyncd.conf"
                      DefineClasses           "hup_inetd"
              }


processes:
      web_master.hup_inetd::
              "inetd" signal=hup inform=true


directories:
      web_master::
              /var/log/cfoutputs   mode=750 owner=daemon group=root inform=false
              /usr/local/bin           mode=755 owner=root group=root inform=false


tidy:
      web_master::
              /var/log/cfoutputs pattern=* age=60 rmdirs=false
```

In this task, we distribute the pre-exec script as well as create the directory where we'll upload the files from clients. We also include a tidy action to remove files older than 60 days from this new directory. The directory will grow without bounds if we don't do this, and a filled disk will surely come back to bite us later.

We don't currently have the tidy action defined in our actionsequence. Let's add it to PROD/inputs/control/cf.control_cfagent_conf, so that it has this for the actionsequence:

```
                    actionsequence = (
                            directories
                            disable
                            packages
                            copy
                            editfiles
                            links
                            files
                            processes
                            shellcommands
                            tidy
                    )
```

To upload the `$workdir/outputs` directory to the central host, create a task at `PROD/inputs/tasks/app/cfengine/cf.upload_cfoutputs_dir` with these contents:

```
control:
        solaris|solarisx86::
                rsync_path       = ( "/opt/csw/bin/rsync" )
        linux::
                rsync_path       = ( "/usr/bin/rsync" )

shellcommands:
        any::
                # the web_master variable holds the name of the host
                # where we run rsync as a daemon
                "$(rsync_path) -a --exclude 'previous' $(workdir)/outputs/ ➥
$(web_master)\:\:outputs-upload/`hostname`"
                        timeout=600 inform=false umask=022
```

Add this task to the `cf.any` hostgroup, so all hosts upload their `outputs` directory on every run.

USING RED HAT AS THE AGGREGATOR HOST

If you run Red Hat Linux on the host where you'd like to aggregate your `outputs` directories, you need to be aware of two things:

- Red Hat uses `xinetd` instead of `inetd`, so `xinetd` will need to be modified to start rsync as a daemon.

- Red Hat runs the rsync daemon as the user nobody, so the permissions on `/var/log/cfoutputs` will need to be changed to be owned by the nobody user.

    We don't cover automation of Red Hat systems for this role, but wanted to point out the obvious modifications in case you want to try it on your own.

We do have a small problem, though. We're missing the rsync program in our base Solaris installation process. We'll want to install it from the Blastwave repository as part of the JumpStart process, as we do for the rest of our open source software additions. Modify the JumpStart `postinstall` script so that rsync is installed by `pkg-get`, by changing this line on *hemingway* (the JumpStart host) in the script `/jumpstart/profiles/aurora/finish_install.sh`:

```
pkg-get install wget gnupg textutils openssl_rt openssl_utils ➥
berkeleydb4 daemontools_core daemontools daemontools_core sudo cfengine subversion
```

We simply want to append rsync to the list:

```
pkg-get install wget gnupg textutils openssl_rt openssl_utils berkeleydb4 ➥
daemontools_core daemontools daemontools_core sudo cfengine subversion rsync
```

Now that the central host *goldmaster* is getting populated with the `outputs` directory output from all clients, you're free to use it as a source for reports. At this early stage in our environment's history, we still like getting the direct e-mail from `cfexecd` from all runs on each host. Once our site grows beyond a few hundred systems running cfengine, we'll probably find it difficult to keep up with and make sense of the e-mails as a whole.

A simple hourly or daily script to summarize and e-mail the aggregated `outputs` directory contents would make more sense at that point. Create a simple script for this purpose at `PROD/repl/admin-scripts/cfoutputs-report` with these contents:

```sh
#!/bin/sh
###########################################################################
# This script only works with GNU find, so make sure it's a Linux host.
###########################################################################
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/opt/admin-scripts

case `hostname` in
goldmaster*)
     echo "This is the proper host on which to run cfoutputs reports, continuing..."
     ;;
*)
     echo "This is NOT the proper host on which run cfoutputs reports, exiting now."
     exit 1
     ;;
esac

THRESHOLD=60
RECIPIENTS=ops@example.org

cd /var/log/cfoutputs && \
for dir in *
do
        find $dir -mmin -$THRESHOLD -type f | xargs cat
done | mail -s"cfoutputs report for last $THRESHOLD minutes" $RECIPIENTS
```

This shell script simply looks for any new files created in the centralized cfoutputs directory during the last 60 minutes (it assumes GNU find is in the path, which is the find command included with Debian GNU/Linux). It outputs the file contents to the mail command using the cat command.

■**Note** The pipe to the mail command is outside the for loop, so we don't get a separate mail for each directory under /var/log/cfoutputs. If you're not sure you understand why this is necessary, try moving the pipe to the mail command to the same line as the find command. Experimenting with shell scripts is one of the best ways to increase your shell-scripting knowledge.

The contents of PROD/repl/admin-scripts/ are already synchronized to all hosts at the location /opt/admin-scripts, so we don't need to make any changes for the script to be copied to the *goldmaster* host. Because this script will be on every host at our site, we make sure that it attempts to run only when invoked on the correct host.

Create a simple task to run the script on an hourly basis, in a file called `PROD/inputs/tasks/app/cfengine/cf.run_cfoutputs_report`, with these contents:

```
shellcommands:
      # since the script needs GNU find, make sure this is a Linux host
      web_master.linux.Min00_05::
             "/opt/admin-scripts/cfoutputs-report"
                    timeout=300 inform=true
```

The `cfoutputs` directory is stored on the host serving the role of `web_master`—because that's where we run the rsync daemon. To have the task used, add this line to the file `PROD/inputs/hostgroups/cf.web_master`:

```
tasks/app/cfengine/cf.run_cfoutputs_report
```

The e-mail output of the script is very basic:

```
From:    root <root@example.org>
To:      ops@example.org
Subject: cfoutputs report for last 60 minutes

cfengine:goldmaster: Executing script /etc/init.d/autofs reload...
(timeout=60,id=-1,gid=-1)
cfengine:goldmaster:t.d/autofs relo: Reloading automounter: checking for changes ...
cfengine:goldmaster:t.d/autofs relo: Reloading automounter map for: /home
cfengine:goldmaster:t.d/autofs relo: Reloading automounter map for: /mnt/pkg
cfengine:goldmaster: Finished script /etc/init.d/autofs reload
cfengine:aurora: Object /usr/dt/bin/dtsession had permission 4555, changed it to 555
cfengine:aurora: Removing setuid (root) flag from /etc/lp/alerts/printer...
cfengine:aurora: Object /etc/lp/alerts/printer had permission 4555,
changed it to 555
```

This is a good way to report on `cfagent` output from `cfexecd`. When you have new reporting needs, you can build on this short example script. Note that `cfexecd` offers this useful feature: it doesn't send a new e-mail when the output of the current run matches that of the previous run. Our example script doesn't implement this functionality; this is left as an exercise for the reader.

When all systems are functioning properly, we should see syslog log messages like this on the cfengine master, regarding all clients:

```
goldmaster cfservd[2004]: Accepting connection from ::ffff:192.168.1.236
```

This lets us know that the host with the IP address `192.168.1.236` is connected to our cfengine master. This is something we expect and require, and if it stops happening, something is wrong. We won't use log reports to tell us if hosts stop contacting the cfengine master; instead, we'll use another method that leverages cfengine itself.

Add a file at the location `PROD/inputs/control/cf.friendstatus` with these contents:

```
control:
        #
        # This fragment, only run on the policy server, generates warnings
        # regarding cfengine clients that have connected before, but have not
        # connected within the past 24 hours (a sign that there is likely a
        # problem with the client in question). Records for clients that have
        # not connected for 14 days are purged from the database, so a down host
        # should generate warnings at that time.
        #
        # Clients unseen for 14 days purged from cf_lastseen.db
        # http://www.cfengine.org/docs/cfengine-Reference.html#lastseenexpireafter
        #
        policyhost::
                LastSeenExpireAfter = ( 14 )

alerts:
        policyhost::
                #
                # Warn about hosts that have not connected within the last X hours
                # http://www.cfengine.org/docs/cfengine-Reference.html#alerts
                #
                FriendStatus(4)
                ifelapsed=60
```

We define a class for the "policyhost" machine because we currently have only a variable by that name (used in copy actions). Add this line to `PROD/inputs/classes/cf.main_classes`:

```
        policyhost              = ( goldmaster )
```

Then import the file into `cfagent.conf` by adding this line:

```
        # alert on missing hosts
        control/cf.friendstatus
```

Now if a host stops contacting the cfengine master for more than four hours, you'll see syslog messages and alerts in the `cfexecd` e-mails from the cfengine master host *goldmaster*.

Doing General syslog Log Analysis

Syslog daemons make it easy to centralize syslog messages. They universally have the ability to forward some or all log messages to other hosts on the network.

We'll use this functionality to send the syslog output from all of our hosts to a single system. Using the syslog-ng open source syslog daemon, we'll store all logs in a directory structure sorted by hostnames and the log-message date. Earlier in the chapter, we set `syslog=on` globally at our site, so we can use syslog log entries to keep track of the actions that cfengine takes. Between the `outputs` directory and the syslog messages, we have a complete history and output of the cfengine activity at our site.

Configuring the syslog Server

We'll set up a new role on our network, that of the syslog loghost. We'll use a DNS alias to refer to the host (instead of using its actual hostname), as well as a role-based class in cfengine to control which host collects all the logs. We'll call the role `sysloghost`, and add a new physical host for this role. All of our Debian hosts are already imaged with the syslog-ng package installed, but we'll need to install a small amount of additional software.

We need to make some additions to FAI on the host *goldmaster* to support this new installation class. Modify `/srv/fai/config/class/60-more-host-classes` so that it has these contents:

```
#! /bin/bash

case $HOSTNAME in
    etchlamp|lamphost|etchlamp*)
        echo "WEB" ;;
    loghost1)
        echo "LOGHOST" ;;
esac
```

Then set up disk partitioning so that it resembles the setup for the `WEB` class, by copying `/srv/fai/config/disk_config/WEB` to `/srv/fai/config/disk_config/LOGHOST`. Next, set up packages for this class in the file `/srv/fai/config/package_config/LOGHOST`:

```
PACKAGES aptitude
sec
ccze
logtail
mailx
```

Add the new system into the DNS, using the IP address `192.168.1.234`. The entry for `PROD/repl/root/etc/bind/debian-ext/db.192.168` is:

```
234        IN      PTR   loghost1.campin.net.
```

Here's the entry for `PROD/repl/root/etc/bind/debian-ext/db.campin.net`:

```
loghost1       IN   A      192.168.1.234
loghost        IN      CNAME    loghost1
sysloghost     IN      CNAME    loghost1
```

Add an entry to `/etc/dhcp3/dhcpd.conf` on *goldmaster* like this (substitute your host's MAC address), and restart `dhcpd`:

```
host loghost1 {hardware ethernet 00:0c:29:09:c1:10;fixed-address loghost1;}
```

Now we need to set up the imaging job in FAI with this command:

```
# fai-chboot -Ifv loghost1
```

Boot the system using PXE, after which FAI will set up the host as our site's syslog server. In 10 or 15 minutes, you can log into the host *loghost1* if you'd like. No need to, though, as we'll do everything from the cfengine master, as usual. You might be noticing that pattern by now.

We'll create a syslog-ng configuration file for the syslog server role first. Place a file at `PROD/repl/root/etc/syslog-ng/syslog-ng.conf-sysloghost` by copying the Debian `/etc/syslog-ng/syslog-ng.conf` file to that location. We first need to make sure that this file has the desired `udp` and `tcp` listen lines enabled. Here's our `source s_all` section, with the additions in bold:

```
source s_all {
        # message generated by Syslog-NG
        internal();
        # standard Linux log source (this is the default place for the syslog()
        # function to send logs to)
        unix-stream("/dev/log");
        # messages from the kernel
        file("/proc/kmsg" log_prefix("kernel: "));
        udp();
        tcp(port(51400) keep-alive(yes) max-connections(30));
};
```

Now that we have syslog-ng on the syslog host configured to listen for syslog connections on the network, we need to add these lines to the end of the file to store the logs in a sorted manner:

```
# set it up
destination std {
file("/var/log/HOSTS/$HOST/$YEAR/$MONTH/$DAY/${FACILITY}_${HOST}_${YEAR}_➡
${MONTH}_${DAY}" owner(root) group(root) perm(0640) dir_perm(0750) ➡
create_dirs(yes));
};

log {
        source(s_all);
        destination(std);
};
```

Now create a file at `PROD/repl/root/etc/syslog-ng/syslog-ng.conf-debian` for all our
Debian hosts, again by copying `/etc/syslog-ng/syslog-ng.conf` to this new file name. Add
these lines to the end of the file:

```
destination loghost {
        tcp("sysloghost.campin.net" port(51400));
};

# send everything to loghost, too
log {
        source(s_all);
        destination(loghost);
};
```

Now we'll create a task for syslog configuration across all systems at our site. Create
a file on the cfengine master at the location `PROD/inputs/tasks/os/cf.configure_syslog`
with these contents:

```
control:
        any::
                addinstallable          = (      hup_syslog_ng
                                                 restart_syslog_ng
                                                 restartsyslogd
                                                 )
                AllowRedefinitionOf     = ( syslog_ng_conf_file )

        debian::
                syslog_ng_conf_file     = ( syslog-ng.conf-debian )

        debian.sysloghost::
                syslog_ng_conf_file     = ( syslog-ng.conf-sysloghost   )
```

```
copy:
      debian::
            $(master_etc)/syslog-ng/$(syslog_ng_conf_file)
                              dest=/etc/syslog-ng/syslog-ng.conf
                              mode=444
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
                              define=hup_syslog_ng

editfiles:
      redhat::
            { /etc/syslog.conf
                  AppendIfNoSuchLine '*.*          @sysloghost.campin.net'
                  DefineClasses "restartsyslogd"
            }

      solaris|solarisx86::
            { /etc/syslog.conf

                  #AutoCreate
                  AppendIfNoSuchLine '*.info              @sysloghost.campin.net'
                  DefineClasses "restartsyslogd"
            }

processes:
      hup_syslog_ng::
            # when syslog-ng.conf is updated, HUP syslog-ng
            "syslog-ng" signal=hup inform=false

      debian::
            "/sbin/syslog-ng" restart "/etc/init.d/syslog-ng start"
                  inform=false umask=022

      restartsyslogd::
            "syslogd" signal=hup inform=true
```

```
shellcommands:
        restart_syslog_ng::
                "/etc/init.d/syslog-ng stop; sleep 1 ; /etc/init.d/syslog-ng start"
                        timeout=10


disable:
        debian::
                # this breaks logrotate, remove them
                /etc/cron.daily/exim4-base
                /etc/logrotate.d/exim4-base
```

In this task, we copy a complete syslog-ng.conf file to Debian hosts, and edit the
syslog.conf file on Solaris and Red Hat to send all syslog messages from the default syslog
daemon. Be sure to insert Tab characters into the syslog.conf file when editing it. Put-
ting Tab characters directly into the editfiles entry will do the right thing. We remove
some logrotate configuration files that get left behind when the Debian postfix package
replaces Exim, the default Debian mail-transfer program. When the logrotate configura-
tion files are in place and the user accounts for Exim are missing (as they are at our site),
the logrotate program fails to run. We remove the files to work around this problem.

Outputting Summary Log Reports

We want to keep a general eye on the syslog messages at our site. Programs like logcheck
compile a summary of the message traffic: they ignore particular messages and display
the rest. This means that over time, we'll have to add messages to an ignore file if we want
to stop seeing them in the reports.

The useful nature of such reports becomes apparent when you see new sorts of mes-
sages that indicate problems or issues. Once you find these issues, you can program them
into your real-time–alerting log tool (such as Simple Event Correlator, or SEC, covered in
the next section) to see them immediately if they recur. The problem is that you have to
learn about the errors first, and that's where logcheck comes in.

We'll utilize newlogcheck, a modification on the logcheck tool that reports from
a central loghost instead of on stand-alone hosts. A second feature of newlogcheck is
that it summarizes entries to report how many times each happened, as opposed to the
logcheck default, which shows each and every message in its entirety.

The first step is to download logcheck from http://sourceforge.net/project/
showfiles.php?group_id=80573&abmode=1. Untar it, and move the systems/linux directory to
PROD/repl/root/usr/pkg/logcheck on your cfengine master. Then download newlogcheck
from http://www.campin.net/download/newlogcheck.tgz and place the newlogcheck.sh and
sort_logs.pl scripts into your logcheck directory. They'll be distributed by cfengine and
ready to run from /usr/pkg/logcheck.

Create a task at PROD/inputs/tasks/app/logcheck/cf.logcheck with these contents:

```
copy:
        sysloghost::
                $(master)/repl/root/usr/pkg/logcheck
                        dest=/usr/pkg/logcheck/
                                        r=inf
                                        mode=750
                                        type=checksum
                                        ignore=/usr/pkg/logcheck/tmp
                                        ignore=tmp
                                        ignore=check
                                        ignore=host
                                        purge=true
                                        server=$(fileserver)
                                        encrypt=true


directories:
        sysloghost::
                /usr/pkg/logcheck/ mode=750 owner=root group=root inform=false


shellcommands:
        #
        # Runnning at 0600 lets us catch the logs before they're rotated.  We
        # then run one more time during the workday to see what's been going
        # on.
        #
        sysloghost.(Hr06|Hr16).Min00_05.weekday::
                "/usr/pkg/logcheck/newlogcheck.sh "
                        timeout=7200 inform=true


        #
        # we rotate logs daily shortly after 6am, so we need a single run on
        # weekend mornings too.
        #
        sysloghost.Hr06.Min00_05.weekend::
                "/usr/pkg/logcheck/newlogcheck.sh "
                        timeout=7200 inform=true
```

Add this line to PROD/inputs/classes/cf.main_classes:

```
sysloghost                = (      loghost1 )
```

Create the hostgroup file for the syslog host role with this file at the location `PROD/inputs/hostgroups/cf.sysloghost`:

```
import:
        any::
                tasks/app/logcheck/cf.logcheck
```

Use this entry to add the new hostgroup into the hostgroup-mappings file `PROD/inputs/hostgroups/cf.hostgroup_mappings`:

```
sysloghost::                     hostgroups/cf.sysloghost
```

This will get logcheck copied to *loghost1*, and it will start running reports at the times listed in the task. You should run it manually once, to make sure it works properly, and you should get a good start on adding entries to the ignore files.

Once you get your first report, note the log entries that you don't want to see in the reports any longer, and note the ignore patterns to the `PROD/repl/root/usr/pkg/logcheck/*ignore` files. The `egrep` command provides the ignore functionality, so the patterns in the files are extended `grep` patterns.

Doing Real-Time Log Reporting

We will utilize our centralized syslog loghost to alert the SA staff when particularly important or notable syslog messages appear. These might be alert messages sent to a pager about DNS server problems or full disks, or they might simply be informational messages sent to the administrator's inbox about how many SSH logins occurred that day.

The Simple Event Correlator (SEC) can do all of that and more. It is an open source program intended to perform event-correlation duties with network-management systems such as HP OpenView or Micromuse (now IBM) OMNIbus. SEC allows you to use messages to set a particular state, and subsequently other states, depending on further events or the passage of time. At any one of these state changes, SEC can send e-mail or execute other commands, or simply drop the current state. (Visit SEC's home page at `http://www.estpak.ee/~risto/sec/`.)

SEC will easily handle our simple log-alerting needs, such as e-mailing when it sees a particular event. It will also handle more advanced reporting such as tracking the process identifier (PID) of a process servicing a particular user's FTP login, recording all log messages from that process, and finally e-mailing all those events when the user logs out.

We will call SEC directly from syslog-ng on the loghost, by piping all messages directly into it. Note that the SEC program itself is located in `/usr/bin` because we installed the Debian SEC package at installation time (using FAI), and that's where Debian places it. First, we'll need a configuration file. Let's place a file at `PROD/repl/root/usr/pkg/sec/etc/sec.conf` with these contents:

```
type=SingleWithSuppress
ptype=RegExp
pattern=([-\w]+)\s+SCSI transport failed: reason 'tran_err': giving up
desc=SCSI giving up error on $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: SCSI errors" ➥
ops@example.org
window=14400
thresh=1

type=single
continue = dontcont
ptype=regexp
pattern=([-\w]+)\s+named: .*CNAME and other data
desc=BIND CNAME error on $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: BIND CNAME errors" ➥
ops@example.org

type=single
continue = dontcont
ptype=regexp
pattern=([-\w]+)\s+named.*rejected due to errors
desc=$0
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: BIND errors" ➥
ops@example.org

type=SingleWithSuppress
ptype=RegExp
pattern=([-\w]+)\s+genunix: .ID \d+ kern.warning. WARNING: Sorry, ➥
no swap space to grow stack for pid
desc=Swap space problem on $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: memory errors" ➥
ops@example.org
window=7200

type=SingleWithSuppress
ptype=RegExp
pattern=([-\w]+)\s+ufs: .ID \d+ kern.notice. NOTICE: alloc: (\S+): file system full
desc=full filesystem $2 on host $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: disk full errors" ➥
ops@example.org
window=7200
```

```
type=SingleWithSuppress
ptype=RegExp
pattern=([-\w]+)\s+postfix.\w+.*: No space left on device
desc=full filesystem reported by postfix on $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: disk full errors" ➥
ops@example.org
window=14400

type=SingleWithSuppress
ptype=RegExp
pattern=([-\w]+)\s+cfengine:.* Clocks differ too much to do copy by date
desc=time problem found by cfengine on $1
action=shellcmd echo $0 | /usr/bin/mail -s"syslog alert: clock sync errors" ➥
ops@example.org
window=3600
```

The `type=single` lines execute an action based on a match—in this case, sending the message to the mail command—and the rule simply ends there. The `type=SingleWithSuppress` line is more interesting, in that it's basically a throttling mechanism. It uses a unique key to identify the uniqueness of the message that it's throttling. The key is composed of the rule file name, the rule ID, and the event-description string derived from the `desc` parameter of the rule definition. This means that if you want to throttle a message across different hosts, or across multiple messages that might differ in fields other than the hostname, you'll need to manipulate the `desc` field in order to normalize the key. Read the SEC man page for more information.

Place a task at `PROD/inputs/tasks/app/sec/cf.sync_sec_config` with these contents to copy the SEC configuration directory:

```
copy:
        any::
                $(master)/repl/root/usr/pkg/sec
                        dest=/usr/pkg/sec
                                        r=inf
                                        mode=755
                                        type=checksum
                                        purge=true
                                        server=$(fileserver)
                                        encrypt=true
                                        define=restart_syslog_ng

directories:
        debian::
                /usr/pkg/sec mode=755 owner=root group=root inform=false
```

Add this task to the `PROD/inputs/hostgroups/cf.sysloghost` file with this entry:

```
tasks/app/sec/cf.sync_sec_config
```

Make the syslog host utilize SEC by adding these lines to `PROD/repl/root/etc/syslog-ng/syslog-ng.conf-sysloghost`:

```
destination d_sec {

        program("/usr/bin/sec -input=\"-\" -conf=/usr/pkg/sec/etc/sec.conf ➥
-intevents -log=/var/log/sec.log");
};

# send all logs to sec
log {
        source(s_all);
        destination(d_sec);
};
```

Now you'll get e-mail alerts when any of those "bad" messages come through. Read the SEC man page to learn more about what it can do, and examine the many example rules that are distributed with the package from the web site.

Seeing the Light

At the start of this chapter, we had an infrastructure with many applications running in it, such as mail, cfengine, Apache, DHCP, DNS, and more, but we had no visibility into any log messages sent from any of those applications.

Now our infrastructure is in good shape with regard to reporting:

- We have regular e-mails summarizing syslog messages from all our systems.

- We have the capability to alert based on particular messages.

- We have alerts from cfengine when clients stop contacting the cfengine master.

- We have full `cfagent` output (as collected by `cfexecd`) automatically pushed to our cfengine master system for troubleshooting and custom reporting.

One very important area where we're still blind: the availability of the network services at our site. We have no automated method of finding out if our web site goes down, if SSH stops working, or if the disks fill up on any of our hosts. We'll address network and host monitoring in the next chapter.

CHAPTER 10

■ ■ ■

Monitoring

We use automation to configure systems and applications in accordance with our wishes. In a perfect world, we would automate all changes to our hosts, and at the end of the workday, we would go home and not have to do any work again until the next morning. SAs will be the first to tell you that the world isn't perfect. Disk drives fill up; power supplies fail; CPUs overheat; software has bugs; and all manner of issues crop up that cause hosts and applications to fail. System and service monitoring is automation's companion tool, used to notify us when systems and applications fail to operate properly.

Right now, we are unaware of any hardware or software problems in our example environment. There are many key failure situations that administrators wish to know about immediately. Some of these are

- When a host is unreachable on the network

- When a network application fails (e.g., the Apache instance for our public web site doesn't serve content properly)

- When an important process is not running on a system

- When resources are overutilized (e.g., high system load, high CPU utilization, and low disk space)

Without an automated mechanism to detect these situations and notify the administrator, problem notification will be performed by users or even customers! The problem (or problems) might have been going on for an extended period of time before the administrator is alerted, which is embarrassing for the administrator and, when reported by a customer, embarrassing for the business as a whole. Clearly, we need a better solution than relying on users or waiting for the administrators to notice anomalies during the normal course of their work.

Aside from immediate errors or failures, we'd like to be aware of general trends in the performance and resource utilization of our systems. We don't want to find out that we lack sufficient CPU capacity for our public web site when the CPU utilization exceeds 95 percent and rarely comes down! Conversely, we don't want to receive an automated notification every time the CPU utilization exceeds a much lower percentage as a sort of early

warning system—this will result in excessive alerts and isn't even an accurate indication of insufficient CPU capacity. Instead, we would like a way to visualize resource utilization trends over time, which allows us to make a prediction about utilization levels in the future. Armed with this information, we can deploy additional systems or install hardware upgrades before they are needed.

An entire software industry exists around automated system and network monitoring, and many open source programs exist as well. We are fortunate that some very high-quality open source monitoring software exists. We will focus on open source monitoring software in this chapter, and we believe that our choices will scale with your environment as it grows.

We have chosen Nagios for our system and service monitoring system. Nagios is flexible and mature, and help is widely available on Internet mailing lists and newsgroups. We feel that we are leaving you in good hands with Nagios once you've completed this chapter.

We have chosen Ganglia for system resource visualization. Ganglia was developed for monitoring large-scale clusters, and we have found it to be very easy to work with. It is also very flexible and should be able to support any custom system graphing required at your site.

As with automation systems, work is always being done on a site's monitoring systems. Applications and hosts are added; applications change and need to be monitored differently; hosts fail permanently; and critical thresholds change. You need to know your monitoring systems inside and out, both the monitoring software itself as well as exactly what is being monitored at your site.

Nagios

Nagios is a system and network monitoring application. It is used to monitor network client/server applications such as POP, HTTP, and SMTP, as well as host resource utilization such as disk and CPU usage. Users usually interact with it though an optional web interface included with the source distribution.

Here is the list of features as documented on `www.nagios.org`:

- Monitoring of network services (SMTP, POP3, HTTP, NNTP, PING, etc.)

- Monitoring of host resources (processor load, disk usage, etc.)

- Simple plug-in design that allows users to easily develop their own service checks

- Parallelized service checks

- Ability to define network host hierarchy using "parent" hosts, allowing detection of and distinction between hosts that are down and those that are unreachable

- Contact notifications when service or host problems occur and get resolved (via e-mail, pager, or user-defined method)

- Ability to define event handlers to be run during service or host events for pro-active problem resolution

- Automatic log file rotation

- Support for implementing redundant monitoring hosts

- Optional web interface for viewing current network status, notification and problem history, log file, and so on

Nagios is widely used and has an active user community. Good support is available on Internet mailing lists and on the `http://www.nagios.org` web site. Also, several books are available on the subject, and one of our favorites is *Building a Monitoring Infrastructure with Nagios* by David Josephsen (Prentice Hall, 2007). We like it because it focuses not just on the Nagios application itself but also on real-world monitoring scenarios. We also recommend *Pro Nagios 2.0* by James Turnbull (Apress, 2006). Both books cover Nagios version 2 but the majority of the content still applies, as does the good advice from each on general monitoring system design.

With many of the applications deployed so far in our example environment, the whirlwind introduction that we provide is enough to give you a good understanding of the software and technologies you're deploying. Nagios is different in that it will definitely require further reading and experimentation on your part. We encourage you to use Nagios, and we provide a working configuration to get it up and running quickly at your site so that you can leverage its feature set. In order to make full use of it, though, you will need to learn more about it on your own.

Nagios Components

Before we go deeply into the configuration of Nagios, we will explain the different parts of the monitoring system that we're going to deploy. The Nagios program itself is only one part of our overall monitoring system. There are four components:

- *The Nagios plug-ins* are utilities designed to be executed by Nagios to report on the status of hosts or services. A standard set of open source plug-ins is available at the `http://www.nagios.org` web site, and many additional plug-ins are freely available on the web. Extending Nagios through the use of custom plug-ins is simple, easy, and encouraged.

- *The Nagios daemon* is a scheduler for plug-ins that perform service and host checks.

- *The web interface* is a set of Common Gateway Interface (CGI) programs that are included in the Nagios source distribution. The CGI interface is completely optional, but it is extremely useful, and we consider it a mandatory part of a complete monitoring infrastructure.

- *Remote plug-in execution via the Nagios Remote Plug-in Executor (NRPE)* is also provided. In order to check local system resources on remote hosts, a mechanism for remote checking is needed. Some sites utilize Simple Network Management Protocol (SNMP) agents for this purpose. The NRPE add-on to Nagios is available at the `http://www.nagios.org` web site and is designed for remote execution of Nagios plug-ins.

We will go into greater depth on the Nagios daemon itself in the next section. First, we want you to be aware that when administrators talk about Nagios, they're usually talking about all the parts that form the complete monitoring system, because every site is required to add plug-ins in order for Nagios to perform service/host checking at all, making for two monitoring system components right away. Most sites also deploy the Nagios web interface, along with a facility to check remote system resources (e.g., NRPE, SNMP, or the Nagios `check_by_ssh` add-on).

At our example site, we'll use Nagios to schedule plug-ins and handle notifications, plug-ins to perform host and service checks, NRPE to run plug-ins on remote hosts, and the Nagios CGI web interface to display status and to interact with users.

Nagios Overview

At its core, Nagios is simply a plug-in scheduling and execution program. The Nagios source distribution itself does not include monitoring scripts or programs, though an open source set of plug-ins is available for download from the Nagios web site. Through the use of plug-ins, Nagios becomes a true monitoring system, as the plug-ins check and report on the availability of hosts and services. Nagios really only understands that exit codes from plug-ins dictate the actions it takes.

Nagios has four types of configuration files:

- The main configuration file is `nagios.conf`. This file contains settings and directives that affect the Nagios daemon itself, as well as the CGI programs. These are settings such as log file locations, whether or not notifications are globally enabled, and so on. This file doesn't directly set up any monitored services.

- Resource file(s) are used to define macros (i.e., strings that are replaced at runtime by the Nagios daemon with the values defined in the resource file). These are used to abstract the locations of plug-ins, as well as to store private information such as passwords. The contents of these files won't be exposed to the CGI programs; therefore, the files can be protected with restrictive file permissions.

- You will spend most of your time in the object definition files. These files define hosts, hostgroups, contacts, services to be monitored, commands used for service and host checks, and more. The sample configuration that we put in place when we build Nagios will automatically configure object definitions to monitor the monitoring host itself. We will add further definitions to monitor the hosts and services on our network.

- The CGI configuration file, obviously, configures the CGI (web interface) programs. It contains a reference to the main configuration file, so that the Nagios daemon configuration and object definition file locations made available to the CGI programs.

The example Nagios configuration from the source distribution (as installed by `make install-config` later in this chapter) uses different files to store different parts of its configuration. We use this same approach in our cfengine configuration files, and it leads to easier comprehension and debugging once the files grow large. We use the `cfg_dir` directive in the main Nagios configuration file (`nagios.conf`) to include all files with a .cfg suffix in a specified directory as object definition files. The `cfg_dir` directive also recursively includes .cfg files in all subdirectories.

Nagios uses templates to implement object inheritance. This allows us to define default settings for objects that can be overridden as needed in host and service definitions. The Nagios example configuration makes use of templates, so we'll already be utilizing them when we get our Nagios installation up and running.

The fundamental building blocks in the Nagios configuration files are host and service definitions. Each is defined as a Nagios object. Host and service definitions are just two examples of the several types of objects.

Here is the complete list of Nagios object types:

- *Hosts*: Hosts contain definitions of hosts or devices. These are usually physical machines, but they might be virtual systems as well. Nagios doesn't know or care about the difference.

- *Hostgroups*: These define one or more hosts and are primarily used to organize and simplify the service definitions, as well as the Nagios web interface.

- *Services*: Service objects define a service running on a host (e.g., the HTTP service running on a web server).

- *Service groups*: These objects define one or more services grouped together and are useful for organizing the Nagios web interface. Service groups are also useful (though still strictly optional) if you later decide to use Nagios notification escalation rules. We won't cover host and service notification escalation in this book, however.

- *Contacts*: This object defines the recipients of notifications (i.e., people to notify).

- *Contact groups*: Contact groups define groups of one or more contacts. Nagios objects that send notifications always reference contact groups.

- *Time periods*: These objects define blocks of time and are used by other definitions to determine when checks are run, notifications are sent, or blackout periods.

- *Commands*: These objects define command macros, which are used to send notifications and execute plug-ins.

CONTACTS VS. CONTACT GROUPS

Be aware that contacts aren't used directly by Nagios when notifications are sent. Service definitions, as well as escalation rules (not covered in this book) utilize contact groups instead. This is probably for the best, since the grouping will allow easy expansion of recipient lists later on. This is not always obvious to new users, however.

The knowledge of Nagios object types, in tandem with the example configuration that we deploy later in this chapter should be enough to get you started with Nagios. Please pick up a good book on Nagios (we recommended two good books earlier in the chapter), join the *nagios-users* mailing list (`nagios-users@lists.sourceforge.net`), and read the online documentation in order to build on your installation from the point where this book leaves off.

Deploying Nagios with cfengine

In this section, we move step by step through the process of building, configuring, and deploying Nagios using automation. We know how hard following such a process can be, so we're attempting to make it as easy on you as possible.

Steps in Deploying Our Nagios Framework

Many steps will be required to deploy a fully functional Nagios framework in an automated fashion. Here are the steps, in order:

1. We add system user accounts to run the Nagios and NRPE programs.

2. We build Nagios from source for Debian i686 and copy the programs to the cfengine master for later automated copying.

3. We build the Nagios plug-ins for Debian i686, Red Hat i686, and Solaris SPARC and manually copy them to the cfengine master for later automated copying to clients.

4. We manually copy the Nagios daemon startup script to the cfengine master for later automated copying to cfengine clients.

5. We separate the Nagios program directory and configuration directory on the cfengine master so that our example site can easily support automated updates of only the Nagios binaries.

6. We manually generate an SSL certificate for the Nagios web interface and copy it to the cfengine master for later automated copying to cfengine clients.

7. We manually create the Apache virtual host configuration on the cfengine master, also for later automated copying to cfengine clients.

8. We create the Nagios web interface authorization file and manually copy it to the cfengine master for later automated copying to cfengine clients.

9. We create a cfengine task to copy files to our site's monitoring host:

   a. The core Nagios programs

   b. The directory containing the Nagios configuration files

   c. The Apache configuration file

   d. The Apache authorization file

   e. The Apache SSL certificate used for the Nagios web interface

10. We configure a Nagios monitoring host role in cfengine using a class.

11. We create a hostgroup file for the new monitoring host role in cfengine.

12. We create a cfengine task to distribute the Nagios plug-ins to all platforms at our example site.

13. We create a hostname in the DNS for the monitoring host.

14. We modify the default localhost-only monitoring done by Nagios so that the monitoring host itself is properly monitored.

15. We build NRPE for Debian i686, Red Hat i686, and Solaris SPARC.

16. We create an NRPE configuration file for our site on the cfengine master, for later automated copying by cfengine clients.

17. We create an NRPE start-up script on the cfengine master, for later automated copying by cfengine clients.

18. We create a cfengine task to copy the NRPE programs, configuration file and start-up script to all hosts.

19. We configure the host-based firewall on Red Hat to allow incoming NRPE connections and then copy the firewall configuration file via a cfengine task.

20. We finally have a complete framework to work with: we add new host and service definitions to the Nagios configuration files on the cfengine master in order to monitor all the hosts at our site.

There are 20 steps required just to set up Nagios! This is probably the most difficult chapter in this book to follow. When describing the cfengine configurations in this chapter, we will focus on the results in regard to setting up Nagios. The actions taken in cfengine should be quite familiar to you by now; they consist mainly of `copy`, `shellcommands`, `links`, and `processes` actions.

Step 1: Creating User Accounts

Using dedicated user accounts for daemons on UNIX systems is good practice. First, you want the daemon to run as a nonroot user so that security vulnerabilities don't grant immediate root privileges to attackers. Second, you want the compromise of one daemon to only affect the files writeable by that user and for any investigation to point quickly back to the daemon at fault. If the same user account is used for many daemons, it could be harder to determine the program that was compromised by an attacker.

We'll use our existing internal web server host named *etchlamp* as our monitoring host. First, create the needed user and groups manually on *etchlamp* (which is running Debian 4.0 on the i686 platform) as the root user:

```
# adduser --system --group --no-create-home nagios
# /usr/sbin/groupadd nagcmd
# /usr/sbin/usermod -G nagcmd nagios
# /usr/sbin/usermod -G nagcmd www-data
```

We added the account file entries for the `nagios` user to the master Debian, Red Hat, and Solaris `shadow`, `passwd`, and `group` files on the cfengine master (in `PROD/repl/root/etc/`), and we added the `nagcmd` group entry to the group files for all three platforms.

Step 2: Building Nagios

The next step was to download the stable version of Nagios from `http://www.nagios.org/download/`. At the time of this writing, the latest stable Nagios version is 3.03. Once downloaded, we had to untar and ungzip it and then build it as follows:

```
# wget http://voxel.dl.sourceforge.net/sourceforge/nagios/nagios-3.0.3.tar.gz
# tar zxf nagios-3.0.3.tar.gz
# cd nagios-3.0.3
# ./configure --with-command-group=nagcmd --prefix=/usr/pkg/nagios-3.0.3
# make all
# make install
# make install-init
# make install-config
# make install-commandmode
# make install-webconf
Password:
/usr/bin/install -c -m 644 sample-config/httpd.conf /etc/apache2/conf.d/nagios.conf

*** Nagios/Apache conf file installed ***
```

Building Nagios is rather easy, and it would be unusual to encounter any errors at build time because of the relative lack of compile-time dependencies. Now that we have built Nagios, we'll need to copy it to our cfengine master for later deployment:

```
# rsync -avze ssh /usr/pkg/nagios-3.0.3/ \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/➥
nagios-3.0.3-debian.i686
```

Step 3: Building the Nagios Plug-ins

Now, we have Nagios compiled, but it won't be useful without plug-ins. To compile the Nagios plug-ins for Debian (i686), we ran these commands on a Debian 4.0 i686 system with a C development environment installed (the latest Nagios plug-ins version at the time of this writing is 1.4.12):

```
# wget \
http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.12.tar.gz
# tar zxf nagios-plugins-1.4.12.tar.gz
# cd nagios-plugins-1.4.12
# ./configure --with-nagios-user=nagios --with-nagios-group=nagios \
--prefix=/usr/pkg/nagios-plugins-1.4.12 && make all &&  make install
```

Then, we copied the programs over to the cfengine master:

```
# rsync -avze ssh --progress /usr/pkg/nagios-plugins-1.4.12/  \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/➥
nagios-plugins-1.4.12-debian.i686
```

So far, we only have the Nagios plug-ins for Debian (i686). We're going to need the plug-ins compiled for all platforms at our site for use with NRPE, covered later in this chapter. To compile the Nagios plug-ins for Red Hat (i686), we ran these commands on the *rhmaster* system, where we have a C development environment:

```
# wget \
http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.12.tar.gz
# tar zxf nagios-plugins-1.4.12.tar.gz
# cd nagios-plugins-1.4.12
# ./configure --with-nagios-user=nagios --with-nagios-group=nagios \
--prefix=/usr/pkg/nagios-plugins-1.4.12 && make all && make install
```

Then, from the cfengine master, we copied the programs:

```
# hostname
goldmaster
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg
# rsync -avze ssh --progress \
rhmaster:/usr/pkg/nagios-plugins-1.4.12/ nagios-plugins-1.4.12-redhat.i686
```

Now, all we have left is Solaris. To compile the Nagios plug-ins for Solaris 10 (SPARC), the procedure is the same, except that the untar and ungzip commands are different:

```
# wget \
http://osdn.dl.sourceforge.net/sourceforge/nagiosplug/nagios-plugins-1.4.12.tar.gz
# gunzip -c nagios-plugins-1.4.12.tar.gz | tar xf -
# cd nagios-plugins-1.4.12
# ./configure --with-nagios-user=nagios --with-nagios-group=nagios \
--prefix=/usr/pkg/nagios-plugins-1.4.12 &&  make all &&  make install
```

Again, we copied the programs over to the cfengine master:

```
# hostname
goldmaster
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg
# rsync -avze ssh --progress --rsync-path=/opt/csw/bin/rsync \
aurora:/usr/pkg/nagios-plugins-1.4.12/ nagios-plugins-1.4.12-sunos.sun4u
```

Step 4: Copying the Nagios Start-up Script on the cfengine Master

Next, we created a directory named PROD/repl/root/init.d on the cfengine master and copied the init script from /etc/init.d/nagios on the system where we built Nagios for

Debian into it (installed by the earlier `make install-init` command that we ran when building Nagios). We'll use cfengine to create the proper links in the `/etc/rcX.d` directories later in this chapter.

Step 5: Separating the Nagios Configuration Directory from the Program Directory

We'll place the Nagios daemon configuration files at `PROD/repl/root/usr/pkg/nagios-conf` by moving the `etc` directory up one directory level and into `nagios-conf`:

```
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg
# mv nagios-3.0.3/etc nagios-conf
# cd nagios-3.0.3/
# ln -s /usr/pkg/nagios-conf etc
```

The directory layout inside `nagios-conf`, as set up by the `make install-config` command in the Nagios source directory (which we ran earlier in the chapter), looks like this:

```
# pwd
/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/nagios-conf
# ls -F
./  ../  cgi.cfg  htpasswd.users  nagios.cfg  objects/  resource.cfg
# ls -F objects/
./  ../  commands.cfg  contacts.cfg  localhost.cfg  printer.cfg switch.cfg
templates.cfg  timeperiods.cfg  windows.cfg
```

Notice the directory named `objects`: it is where the example configuration places all the Nagios objects used to configure monitored hosts and services. We'll continue to use this directory for the objects that we define.

While we're here in the Nagios configuration file directory, we'll modify the `resource.cfg` file. We need to change the `$USER1$` line from this:

```
# Sets $USER1$ to be the path to the plug-ins
$USER1$=/usr/pkg/nagios-3.0.3/libexec
```

to this:

```
# Sets $USER1$ to be the path to the plug-ins
$USER1$=/usr/pkg/nagios-plugins/libexec
```

The `$USER1$` macro sets the location of our Nagios plug-ins. We installed them to a directory outside of the main Nagios directory in step three, so we need to have Nagios look for them in the new directory.

Create the Nagios Web Interface Configuration Files

In this section, we'll cover steps six through eight, which are the creation of Apache configuration and authorization files and an SSL certificate.

Step 6: Generating an SSL Certificate for the Nagios Web Interface

We generated the SSL certificate for our Nagios web interface with this command (as the root user on the host *etchlamp*):

```
# /usr/sbin/make-ssl-cert /usr/share/ssl-cert/ssleay.cnf /etc/apache2/ssl/nagios.pem
# cd /etc/apache/ssl
# scp 5796a599 nagios.pem \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/apache2/ssl/
```

Note that, after generating the certificate, we copied it to the cfengine master. We will automate the distribution of this file using cfengine, as usual.

Step 7: Creating the Apache VirtualHost Configuration for the Nagios Web Interface

We'll need to configure Apache with the required directives to serve our Nagios web interface using the Nagios CGI programs. In addition, we need to make sure that authentication is used, since the Nagios web interface contains information we only want authorized staff to view and modify.

Modification operations are those that stop alerts for some or all systems, send manual alerts, or manually change the status of a host check. We want to protect the integrity of our Nagios framework by controlling access.

Here is our Apache configuration file, which we've placed on the cfengine master at the location `PROD/repl/root/etc/apache2/sites-available/nagios.conf`:

```
NameVirtualHost *:443

<VirtualHost *:443>
        ServerName    nagios.campin.net
        ServerAlias     nagios

        DocumentRoot  /var/www/
        ServerAdmin   admins@example.org
        ErrorLog      /var/log/apache2/nagios-error.log
        CustomLog     /var/log/apache2/nagios-access.log combined
```

```
        SSLEngine on
        SSLCertificateFile /etc/apache2/ssl/nagios.pem

        ScriptAlias /nagios/cgi-bin "/usr/pkg/nagios/sbin"

        <Directory "/usr/pkg/nagios/sbin">
                SSLRequireSSL
                Options ExecCGI
                AllowOverride None
                Order allow,deny
                Allow from all
                AuthName "Nagios Access"
                AuthType Basic
                AuthUserFile /usr/pkg/nagios/etc/htpasswd.users
                Require valid-user
        </Directory>

        Alias /nagios "/usr/pkg/nagios/share"

        <Directory "/usr/pkg/nagios/share">
                SSLRequireSSL
                Options None
                AllowOverride None
                Order allow,deny
                Allow from all
                AuthName "Nagios Access"
                AuthType Basic
                AuthUserFile /usr/pkg/nagios/etc/htpasswd.users
                Require valid-user
        </Directory>
</VirtualHost>
```

We have been avoiding in-depth explanations of Apache configuration files, and we continue the trend here. Just be aware that you shouldn't remove the authentication requirements if you have trouble making user accounts work. Take the time to do it right. Protecting your monitoring web interface from unauthorized access is important.

Step 8: Create the Nagios Web Interface Authentication File

We created the Apache user authentication file on the same system (*etchlamp*), and copied it to the PROD/repl/root/usr/pkg/nagios-conf/ directory on the cfengine master:

```
# htpasswd -c /usr/pkg/nagios/etc/htpasswd.users nagiosadmin
New password:
Re-type new password:
Adding password for user nagiosadmin
# scp /usr/pkg/nagios/etc/htpasswd.users \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/nagios-conf/
root@goldmaster's password:
htpasswd.users                              100%   26     0.0KB/s   00:00
```

The `nagiosadmin` user is special, in that it will have all the required access to the
Nagios web interface that you will require. *Always* create this user account.

Once you have Nagios up and running properly, read the online Nagios authentica-
tion documentation at the URL `http://nagios.sourceforge.net/docs/3_0/cgiauth.html` to
learn to configure additional users.

Step 9: Copying the Nagios Daemon and Configuration Files with cfengine

In steps one through eight, we put together all the building blocks to set up a working
Nagios instance. The bare minimum setup is in place:

- The Nagios daemon

- The Nagios plug-ins

- A web interface

We don't yet have everything that we will want in our final framework, but we do
have everything that we need to automate the copy and setup of Nagios and the Nagios
web interface to our monitoring server. We'll set that up now in cfengine.

First, we created the directory `PROD/inputs/tasks/app/nagios` on the cfengine master
and put a task named `cf.nagios_sync` into it with these contents (explained section by
section):

```
control:
        nagios_host::
               nagios_ver      = ( "nagios-3.0.3" )
      addinstallable = ( restart_nagios  restart_apache2 setup_nagios_rc_scripts )
```

First, we define a variable containing our current Nagios version. Using a variable in
all the places that the version-specific directory name is needed will make it much easier
to upgrade Nagios in the future. We'll only need to build the new version, place it on the
cfengine master and update the variable in this task:

```
classes:  # synonym groups:
        nagios_host.i686::
                have_nagios_dir     = ( "/usr/bin/test -d /usr/pkg/$(nagios_ver)" )
```

Here, we set up a class based on the existence of the current Nagios directory to be used in the next section.

```
copy:
        nagios_host.debian.i686.!have_nagios_dir::
                $(master)/repl/root/usr/pkg/$(nagios_ver)-debian.i686
                                dest=/usr/pkg/$(nagios_ver)
                                r=inf
                                mode=755
                                owner=nagios
                                group=nagios
                                ignore=etc
                                exclude=etc
                                ignore=rw
                                exclude=rw
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_nagios
```

Here, we copy the Nagios programs when the directory meant to hold the current Nagios version doesn't already exist. This is done with the bang (!) class negation operator (i.e., !have_nagios_dir). The Nagios binaries were built for the Debian i686 platform, so we also make use of the debian and i686 classes to make sure that we only copy the binaries to the correct platform:

```
        nagios_host.debian::
                $(master)/repl/root/usr/pkg/nagios-conf
                                dest=/usr/pkg/nagios-conf
                                mode=644
                                r=inf
                                owner=nagios
                                group=nagios
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_nagios
                                purge=true
```

Next, we copy the entire `nagios-conf` directory from the master to the client. We may end up deploying several versions of Nagios at once, but we'd like the path to the configuration files should always remain constant. This is easy to ensure when the configuration files are maintained separately from the programs themselves:

```
$(master)/repl/root/etc/init.d/nagios
                    dest=/etc/init.d/nagios
                    mode=755
                    owner=root
                    group=root
                    type=checksum
                    server=$(fileserver)
                    encrypt=true
                    define=setup_nagios_rc_scripts
                    define=restart_nagios
```

After that, we copy the Nagios startup script into place. Later in this task, we'll create the proper symlinks in the `/etc/rc?.d/` directories. Notice that two different classes are defined when the init script is copied into place. Both trigger actions later in the task, one is meant to restart Nagios since new start-up options may be in use (`restart_nagios`), and the other is meant to ensure that the start-up script symlinks are properly created (`setup_nagios_rc_scripts`):

```
$(master_etc)/apache2/sites-available/nagios.conf
                    dest=/etc/apache2/sites-available/nagios.conf
                    mode=444
                    owner=root
                    group=root
                    type=checksum
                    server=$(fileserver)
                    encrypt=true
                    define=restart_apache2

$(master_etc)/apache2/ssl/nagios.pem
                    dest=/etc/apache2/ssl/nagios.pem
                    mode=444
                    owner=root
                    group=root
                    type=checksum
                    server=$(fileserver)
                    encrypt=true
                    define=restart_apache2
```

```
$(master_etc)/apache2/ssl/5796a599
                        dest=/etc/apache2/ssl/5796a599
                        mode=444
                        owner=root
                        group=root
                        type=checksum
                        server=$(fileserver)
                        encrypt=true
                        define=restart_apache2
```

The three copies in the preceding code are used to place web interface files in place: the VirtualHost configuration for our Nagios web site and the SSL certificate we generated for *nagios.campin.net*:

```
shellcommands:
        debian.nagios_host.restart_nagios::
                "/etc/init.d/nagios restart" timeout=60 inform=true


        debian.nagios_host.restart_apache2::
                "/etc/init.d/apache2 restart" timeout=60 umask=022
```

The preceding restarts are triggered when configuration file or program file updates are made in earlier copy sections. We always want to put new configurations or programs into immediate use, and these shellcommands take care of that for us:

```
        debian.nagios_host.setup_nagios_rc_scripts::
                # This is really only needed the first time Nagios is setup,
                # but this is a totally non-destructive command if run when the
                # links are already there. We are safe.
                "/usr/sbin/update-rc.d nagios start 30 2 3 4 5 . stop 70 0 1 6 ."
                        timeout=60 umask=022
```

The preceding section bears a little explanation. We call the Debian update-rc.d utility which is used to create links in the /etc/rc?.d directories. We could add a list of symlinks to create in the cfengine configuration, but frankly, this is easier. The rest of the task follows:

```
directories:
        debian.i686.nagios_host::
                /usr/pkg/$(nagios_ver)/var/rw m=2775 owner=nagios
                        group=nagcmd inform=false
```

Here, we create a directory used by Nagios to store state information. It is critical that the ownership of the directory and permissions allow the user running the Nagios daemon to write files in it. We use cfengine to regularly ensure that this is the case.

```
processes:
        debian.nagios_host::

                "nagios" restart "/etc/init.d/nagios start" inform=true umask=022

                "/usr/sbin/apache2" restart "/etc/init.d/apache2 start"
                                            inform=true umask=022
```

These are simple process monitors that cause Apache and Nagios to be started up if they're not running on the nagios_host system. We'll define that class in cfengine in step ten.

```
links:
        debian.nagios_host::
                /usr/pkg/$(nagios_ver)/etc    ->! /usr/pkg/nagios-conf
                /usr/pkg/nagios                      ->! /usr/pkg/$(nagios_ver)
                /etc/apache2/sites-enabled/nagios.conf ->!
                                        /etc/apache2/sites-available/nagios.conf

                # the make-ssl-cert utility created this link when we created
                # nagios.pem, we'll preserve it using cfengine
                /etc/apache2/ssl/5796a599       ->! /etc/apache2/ssl/nagios.pem
```

This is the end of the cf.nagios_sync cfengine task. Notice that we're careful to copy the i686 binaries *only* to appropriate hosts by specifying the i686 class in the copy. It obviously wouldn't do any good to copy i686 Linux binaries to a Solaris SPARC system or a Debian x86-64 system (one without compatibility libraries), so we are defensive in our cfengine tasks and allow copies to happen only when conditions exactly match what we are expecting.

In the copy action in the preceding task, we copy the Nagios binaries only when the /usr/pkg/nagios-3.0.3 directory doesn't exist. We don't think there's any reason to regularly sync the files. If you're worried about something outside of cfengine changing those files, you could remove the !have_nagios_dir portion from the copy action and always enforce the proper directory contents.

Step 10: Configuring a Nagios Monitoring Host Role in cfengine

We're making the host *etchlamp* the `nagios_host` machine mentioned in the task in step nine, and to set it, we added this line to `PROD/inputs/classes/cf.main_classes`:

```
nagios_host             = ( etchlamp )
```

Step 11: Creating a Hostgroup File for the Monitoring Host Role in cfengine

To complete our Nagios role configuration in cfengine, we added this line to `PROD/inputs/hostgroups/cf.hostgroup_mappings`:

```
nagios_host::                    hostgroups/cf.nagios_host
```

Then, we created a file on the cfengine master at the location `PROD/inputs/hostgroups/cf.nagios_host` with these contents:

```
import:
        any::
                tasks/app/nagios/cf.nagios_sync
```

Step 12: Copying the Nagios Plug-ins with cfengine

We will handle the distribution of the Nagios plug-ins in a second task, which we will now describe. We created a task on the cfengine master at the location `PROD/inputs/tasks/app/nagios/cf.nagios_plugins_sync` with these contents (explained section by section):

```
control:
        any::
                plugins_ver    = ( "nagios-plugins-1.4.12" )
```

As we did in `cf.nagios_sync`, we use a variable to contain the version-specific directory name, which makes it extremely easy to deploy updates later on but still keep a copy of the previous build.

The rest of this task simply copies the proper plug-in binaries to each platform at our site and has special single file copies that enforce the setuid bit on binaries that require it (for execution with elevated system privileges).

```
copy:
        redhat.i686::
                $(master)/repl/root/usr/pkg/$(plugins_ver)-redhat.i686
                                dest=/usr/pkg/$(plugins_ver)
                                r=inf
                                mode=755
                                owner=nagios
                                group=nagios
                                ignore=rw
                                exclude=rw
                                type=checksum
                                purge=false
                                server=$(fileserver)
                                encrypt=true
                                exclude=check_dhcp
                                exclude=check_icmp

        $(master)/repl/root/usr/pkg/$(plugins_ver)-redhat.i686/libexec/check_dhcp
                                dest=/usr/pkg/$(plugins_ver)/libexec/check_dhcp
                                mode=4555
                                type=checksum
                                server=$(fileserver)
                                trustkey=true
                                encrypt=true
                                owner=root
                                group=root

        $(master)/repl/root/usr/pkg/$(plugins_ver)-redhat.i686/libexec/check_icmp
                                dest=/usr/pkg/$(plugins_ver)/libexec/check_icmp
                                mode=4555
                                type=checksum
                                server=$(fileserver)
                                trustkey=true
                                encrypt=true
                                owner=root
                                group=root
```

```
debian.i686::
        $(master)/repl/root/usr/pkg/$(plugins_ver)-debian.i686
                        dest=/usr/pkg/$(plugins_ver)
                        r=inf
                        mode=755
                        owner=nagios
                        group=nagios
                        ignore=rw
                        exclude=rw
                        type=checksum
                        purge=false
                        server=$(fileserver)
                        encrypt=true
                        exclude=check_dhcp
                        exclude=check_icmp

    $(master)/repl/root/usr/pkg/$(plugins_ver)-debian.i686/libexec/check_dhcp
                        dest=/usr/pkg/$(plugins_ver)/libexec/check_dhcp
                        mode=4555
                        type=checksum
                        server=$(fileserver)
                        trustkey=true
                        encrypt=true
                        owner=root
                        group=root

    $(master)/repl/root/usr/pkg/$(plugins_ver)-debian.i686/libexec/check_icmp
                        dest=/usr/pkg/$(plugins_ver)/libexec/check_icmp
                        mode=4555
                        type=checksum
                        server=$(fileserver)
                        trustkey=true
                        encrypt=true
                        owner=root
                        group=root
```

```
sunos_sun4u::
        $(master)/repl/root/usr/pkg/$(plugins_ver)-sunos.sun4u
                        dest=/usr/pkg/$(plugins_ver)
                        r=inf
                        mode=755
                        owner=nagios
                        group=nagios
                        ignore=rw
                        exclude=rw
                        type=checksum
                        purge=false
                        server=$(fileserver)
                        encrypt=true
                        exclude=check_dhcp
                        exclude=check_icmp
                        exclude=pst3

    $(master)/repl/root/usr/pkg/$(plugins_ver)-sunos.sun4u/libexec/check_dhcp
                        dest=/usr/pkg/$(plugins_ver)/libexec/check_dhcp
                        mode=4555
                        type=checksum
                        server=$(fileserver)
                        trustkey=true
                        encrypt=true
                        owner=root
                        group=root

    $(master)/repl/root/usr/pkg/$(plugins_ver)-sunos.sun4u/libexec/check_icmp
                        dest=/usr/pkg/$(plugins_ver)/libexec/check_icmp
                        mode=4555
                        type=checksum
                        server=$(fileserver)
                        trustkey=true
                        encrypt=true
                        owner=root
                        group=root
```

```
                    $(master)/repl/root/usr/pkg/$(plugins_ver)-sunos.sun4u/libexec/pst3
                              dest=/usr/pkg/$(plugins_ver)/libexec/pst3
                              mode=4555
                              type=checksum
                              server=$(fileserver)
                              trustkey=true
                              encrypt=true
                              owner=root
                              group=root


links:
      any::
              /usr/pkg/nagios-plugins ->! /usr/pkg/$(plugins_ver)
```

We use the plugins_ver variable in this task to create version-specific directories on our hosts and have cfengine create a symlink so that we always have a single filesystem path to our current installation (so that the path /usr/pkg/nagios-plugins will always work).

We want the plug-ins installed on all of our hosts, so we added this task to PROD/ inputs/hostgroups/cf.any with this line:

```
tasks/app/nagios/cf.nagios_plugins_sync
```

Step 13: Creating a DNS Entry for the Monitoring Host

We then added an alias to our DNS so that we can use the hostname *nagios.campin.net* when accessing the Nagios server. Using the alias will allow us to easily migrate Nagios to another server in the future without any users noticing or needing to access a new URL. In order for this to be effective, we need to be sure to only give out the URL *https://nagios. campin.net* and never refer to the system's real hostname.

To create the DNS alias, we added this line to PROD/repl/root/etc/bind/ debian-ext/db.campin.net (and of course, we incremented the zone's serial number and ran named-checkzone):

```
nagios        300     IN     CNAME   etchlamp
```

Once cfengine ran again (according to the schedule defined for cfexecd), we visited the URL *https://nagios.campin.net* in a web browser. We were pleased to be prompted to log into Nagios with a username/password prompt. We used the nagiosadmin account we created, and we were presented with the Nagios web interface.

If you click Service Detail in the left-hand frame, you'll see details for the system "localhost" in the right-hand frame. It should look like this screenshot:
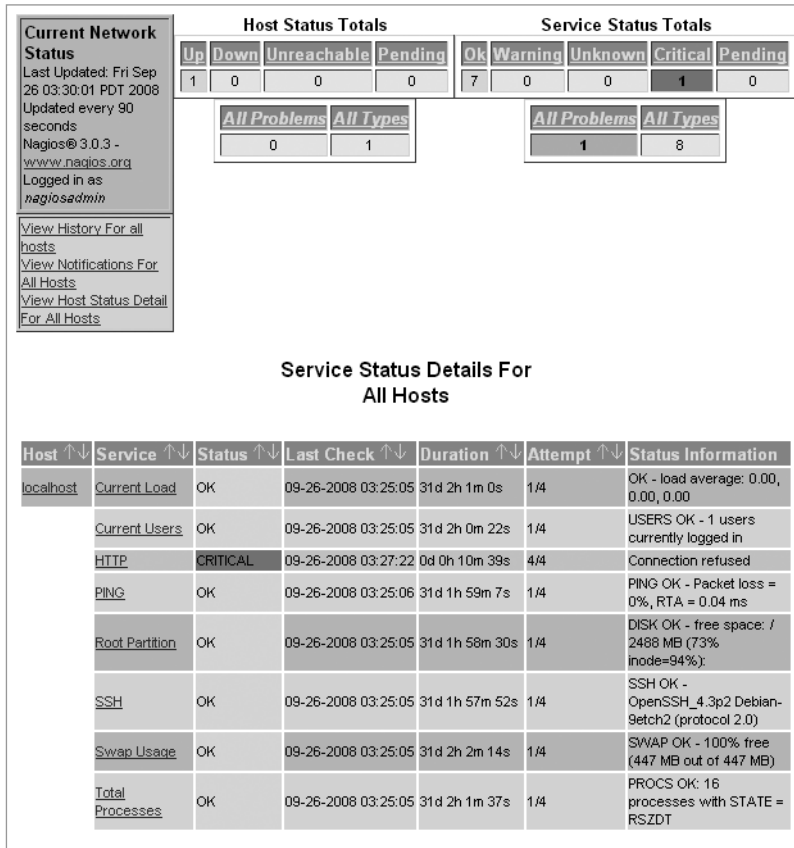
Figure 10-1. *Nagios service detail screen for the system localhost*

By default, Nagios assumes that the standard plug-ins are installed (which is true in our case), and it has an object configuration file called localhost.cfg that sets up the checks you see on that page.

Note that there is a failed check (CRITICAL state) for the HTTP service, because we only run an SSL-enabled Apache server at our site for Nagios, and we have no HTTP service at all. We'll take steps to make Nagios monitor the correct service in step 14.

Step 14: Modifying the Nagios Localhost-Only Monitoring to Check HTTPS

The only system monitored at this point is the actual host running Nagios (*etchlamp*), since only the host localhost has checks defined in the default Nagios configuration files.

We had to change these lines in PROD/repl/root/usr/pkg/nagios-conf/objects/localhost.cfg in order to properly monitor HTTPS on this host:

```
define service{
        use                             local-service    ; service template to use
        host_name                       localhost
        service_description             HTTP
        check_command                   check_http
        notifications_enabled            0
        }
```

We changed them to this:

```
define service{
        use                             local-service ; service template to use
        host_name                       localhost
        service_description             HTTPS
        check_command                   check_https!443!/
        notifications_enabled            0
        }
```

If you're following along with the book in an environment of your own, you'll notice a problem—there isn't a check_https command definition. We had to create it with this addition to PROD/repl/root/usr/pkg/nagios-conf/objects/commands.cfg:

```
define command{
        command_name    check_https
        command_line    $USER1$/check_http -S -L -H $HOSTNAME$ -p $ARG1$ -u $ARG2$
        }
```

This new command object definition calls the check_http plug-in with the appropriate arguments to test an HTTPS-enabled web site. Once this was copied to our Nagios server and Nagios automatically restarted (by cfengine), the proper command was executed and the check cleared in Nagios.

Nagios is now in a fully functional state in our environment, but we don't find it very useful to only monitor a single machine. Next, we'll take steps to monitor the rest of the hosts at our site. The first step will be to deploy a local monitoring agent called NRPE to all our systems.

NRPE

NRPE is the Nagios Remote Plug-in Executor. It is used in place of agents and protocols such as SNMP for remotely monitoring hosts. It grants access to remote hosts to execute plug-ins such as those in the Nagios plug-ins distribution. NRPE has two components: a daemon called nrpe and a plug-in to the Nagios daemon called check_nrpe.

The NRPE documentation points out that there are other ways to accomplish remote plug-in execution, such as the Nagios `check_by_ssh` plug-in. While SSH access to a remote host seems attractive for security reasons, it imposes more overhead on remote hosts than the NRPE program does. In addition, a site's security policy may expressly forbid remote login access by accounts not owned by a real person. We like NRPE because it is lightweight, flexible, and fast.

Step 15: Building NRPE

The NRPE source distribution does not include an installation facility. Once it is built, it is up to us to install it properly, which we'll handle with cfengine. We will build the NRPE binaries and place them on the cfengine master for distribution to all our hosts. We created a single new directory under `PROD/repl/root/usr/pkg` to house the NRPE binaries for each of our platforms: `PROD/repl/root/usr/pkg/nrpe-2.12-bin`.

Now we need to build NRPE so that we have something to place in this new directory. We used these commands to download and compile the program on Debian (i686):

```
# wget http://internap.dl.sourceforge.net/sourceforge/nagios/nrpe-2.12.tar.gz
# gunzip -c nrpe-2.12.tar.gz | tar xf -
# cd nrpe-2.12
# ./configure --enable-ssl && make all
# cd src
# scp nrpe goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/➥
nrpe-2.12-bin/nrpe-debian.i686
# scp check_nrpe goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/➥
pkg/nagios-plugins-1.4.12-debian.i686/libexec/
```

We copied `check_nrpe` to the preexisting `nagios-plugins` directory for the `debian.i686` architecture and copied the `nrpe` program itself into the single shared `PROD/repl/root/usr/pkg/nrpe-2.12-bin` directory.

To build on our Red Hat i686 systems, the commands were the same as for Debian, except that we copied the plug-ins to the `nrpe-bin/nrpe-redhat.i686` directory and the `nrpe` binary to `nrpe-2.12-bin/nrpe-redhat.i686`.

To build NRPE on Solaris, we had to comment out lines 616 though 619 in `nrpe.c`, because the code assumes that all UNIX-like systems have the same syslog facilities as Linux (and Solaris doesn't).

```
616  /*      else if(!strcmp(varvalue,"authpriv"))
617                  log_facility=LOG_AUTHPRIV;
618          else if(!strcmp(varvalue,"ftp"))
619                  log_facility=LOG_FTP; */
```

After that we were able to build on Solaris 10 and copy the programs to the cfengine master with these commands:

```
# ./configure --enable-ssl --with-ssl=/usr/sfw --with-ssl-lib=/usr/sfw/lib && \
make all
# cd src
# scp nrpe goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/➥
nrpe-2.12-bin/nrpe-sunos_sun4u
# scp check_nrpe goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/➥
usr/pkg/nagios-plugins-1.4.12-sunos.sun4u/libexec/
```

The preceding `configure` line makes NRPE compile against the Solaris 10 OpenSSL libraries. We then placed the two resulting binaries into directories on the cfengine master as shown.

Step 16: Creating an NRPE Configuration File

We copied the sample NRPE configuration from the source distribution (in `sample-con-fig/nrpe.cfg`) to the cfengine master at `PROD/repl/root/usr/pkg/nrpe-conf/nrpe.cfg`. We then edited the `nrpe.cfg` file to use the `/usr/pkg/nagios-plugins/libexec` directory for all the paths and allow access from our *etchlamp* system as shown:

```
# substitute your monitoring host's IP for 192.168.1.239
allowed_hosts=127.0.0.1,192.168.1.239

# The following examples use hardcoded command arguments...
command[check_users]=/usr/pkg/nagios-plugins/libexec/check_users -w 5 -c 10
command[check_load]=/usr/pkg/nagios-plugins/libexec/check_load -w 15,10,5 ➥
-c 30,25,20
command[check_hda1]=/usr/pkg/nagios-plugins/libexec/check_disk -w 20% ➥
-c 10% -p /dev/hda1
command[check_zombie_procs]=/usr/pkg/nagios-plugins/libexec/check_procs -w 5 ➥
-c 10 -s Z
command[check_total_procs]=/usr/pkg/nagios-plugins/libexec/check_procs -w 150 -c 200
```

At this point, we have the NRPE programs built and ready for distribution from the cfengine master, along with a configuration file. The last thing we need to prepare for NRPE is a start-up script.

Step 17: Creating an NRPE Start-up Script

We created a simple init script for NRPE at `PROD/repl/root/etc/init.d/nrpe` on the cfengine master with these contents:

```
#!/bin/sh
PATH=/bin:/usr/bin

case "$1" in
start)
        /usr/pkg/nrpe/sbin/nrpe -c /usr/pkg/nrpe/etc/nrpe.cfg -d
        ;;
restart)
        kill `cat /var/run/nrpe.pid`
        pkill -9 -f "nrpe -c /usr/pkg/nrpe/etc/nrpe.cfg -d"
        /usr/pkg/nrpe/sbin/nrpe -c /usr/pkg/nrpe/etc/nrpe.cfg -d
        ;;
stop)
        kill `cat /var/run/nrpe.pid`
        pkill -9 -f "nrpe -c /usr/pkg/nrpe/etc/nrpe.cfg -d"
        ;;
*)
        echo "Usage: $0 {start|stop|restart}"
        exit 1
        ;;
esac

exit 0
```

This is a very simple init script, but it suffices because NRPE is a very simple daemon. We added the `pkill` command, because in writing this chapter, we found that occasionally the PID of the `nrpe` process wasn't properly stored in the `nrpe.pid` file. Occasionally, daemons have bugs such as this, so we simply work around it with some extra measures to kill the daemon with the `pkill` command.

Step 18: Copying NRPE Using cfengine

We now have everything we need to deploy NRPE at our site. To distribute NRPE with cfengine, we created a task to distribute the configuration file, init script, and binaries in a file named `PROD/inputs/tasks/app/nagios/cf.nrpe_sync`. Here's the file, which we will describe only briefly after showing the complete contents, because we're not introducing any new cfengine functionality in this task:

```
control:
        any::
                addinstallable  = ( restart_nrpe  )
                nrpe_ver        = ( "nrpe-2.12" )


copy:
        debian.i686::
                $(master)/repl/root/usr/pkg/$(nrpe_ver)-bin/nrpe-debian.i686
                                dest=/usr/pkg/$(nrpe_ver)/sbin/nrpe
                                mode=755
                                owner=nagios
                                group=nagios
                                ignore=rw
                                exclude=rw
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_nrpe


        redhat.i686::
                $(master)/repl/root/usr/pkg/$(nrpe_ver)-bin/nrpe-redhat.i686
                                dest=/usr/pkg/$(nrpe_ver)/sbin/nrpe
                                mode=755
                                owner=nagios
                                group=nagios
                                ignore=rw
                                exclude=rw
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_nrpe


        sunos_sun4u::
                $(master)/repl/root/usr/pkg/$(nrpe_ver)-bin/nrpe-sunos.sun4u
                                dest=/usr/pkg/$(nrpe_ver)/sbin/nrpe
                                mode=755
                                owner=nagios
                                group=nagios
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
                                define=restart_nrpe
```

```
            any::
                    $(master)/repl/root/etc/init.d/nrpe
                                      dest=/etc/init.d/nrpe
                                      mode=755
                                      owner=root
                                      group=root
                                      type=checksum
                                      server=$(fileserver)
                                      encrypt=true
                                      define=restart_nrpe

                    $(master)/repl/root/usr/pkg/nrpe-conf/nrpe.cfg
                                      dest=/usr/pkg/nrpe/etc/nrpe.cfg
                                      mode=755
                                      owner=nagios
                                      group=nagios
                                      ignore=rw
                                      exclude=rw
                                      type=checksum
                                      server=$(fileserver)
                                      encrypt=true
                                      define=restart_nrpe

shellcommands:
        restart_nrpe::
                "/etc/init.d/nrpe restart" timeout=60 inform=true

directories:
        any::
                /usr/pkg/$(nrpe_ver)/sbin m=775 owner=nagios
                                                group=nagcmd inform=false

                /usr/pkg/$(nrpe_ver)/etc m=775 owner=nagios
                                                group=nagcmd inform=false

processes:
        any::
                "nrpe" restart "/etc/init.d/nrpe start" inform=true umask=022
```

```
links:
      any::
              /usr/pkg/nrpe                   ->! /usr/pkg/$(nrpe_ver)

              # rc scripts
              /etc/rc0.d/K02nrpe     ->! /etc/init.d/nrpe
              /etc/rc1.d/K02nrpe     ->! /etc/init.d/nrpe
              /etc/rc2.d/S98nrpe     ->! /etc/init.d/nrpe

      any.!(solaris|solarisx86)::
              /etc/rc3.d/S98nrpe     ->! /etc/init.d/nrpe
              /etc/rc4.d/S98nrpe     ->! /etc/init.d/nrpe
              /etc/rc5.d/S98nrpe     ->! /etc/init.d/nrpe
              /etc/rc6.d/K02nrpe     ->! /etc/init.d/nrpe
```

When we link the /etc/init.d/nrpe start-up script into the runlevel-specific directories in the preceding links section, we avoid creating a link in /etc/rc3.d on Solaris hosts. This is because Solaris executes the scripts starting with a capital "S" in the directories /etc/rc2.d/ and /etc/rc3.d/ when booting into runlevel 3. We don't want the script to execute twice. No damage would result, but we don't want to be sloppy. Furthermore, the directories rc4.d, rc5.d, and rc6.d don't exist on Solaris, so we won't attempt to create symlinks in them.

Note that we make it easy to move to a newer version of NRPE later on, using version numbers and a symlink at /usr/pkg/nrpe to point to the current version. The use of a variable means only the single entry in this task will need to change once a new NRPE version is built and placed in the appropriate directories on the cfengine master.

To activate this new task, we placed the following line in PROD/inputs/hostgroups/ cf.any:

```
tasks/app/nagios/cf.nrpe_sync
```

Step 19: Configuring the Red Hat Local Firewall to Allow NRPE

The next-to-last step we had to take was to allow NRPE connections through the Red Hat firewall. To do so, we added rules directly to the /etc/sysconfig/iptables file on the system *rhlamp* and restarted iptables with service iptables restart. Here are the complete contents of the iptables file, with the newly added line in bold:

```
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:RH-Firewall-1-INPUT - [0:0]
-A INPUT -j RH-Firewall-1-INPUT
-A FORWARD -j RH-Firewall-1-INPUT
-A RH-Firewall-1-INPUT -i lo -j ACCEPT
-A RH-Firewall-1-INPUT -p icmp --icmp-type any -j ACCEPT
-A RH-Firewall-1-INPUT -p 50 -j ACCEPT
-A RH-Firewall-1-INPUT -p 51 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp --dport 5353 -d 224.0.0.251 -j ACCEPT
-A RH-Firewall-1-INPUT -p udp -m udp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -p tcp -m tcp --dport 631 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 443 -j ACCEPT
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 5666 -j ACCEPT
-A RH-Firewall-1-INPUT -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

When this change started allowing connections to our local NRPE daemon, we decided to enforce the contents of this file using cfengine. This decision will disallow the future use of utilities such as system-config-securitylevel to manage the host's firewall rules, but that's good. Stringent enforcement of the iptables file contents will force the firewall rules to be configured according to our wishes every time cfengine runs. We can always use the Red Hat command system-config-securitylevel to make changes and then feed the resulting /etc/sysconfig/iptables changes back into the copy that we distribute with cfengine. This is just another example of how manual changes are often needed to determine how to automate something. It's always OK as long as we feed the resulting changes and steps back into cfengine for long-term enforcement.

We placed the iptables file on our cfengine master at PROD/repl/root/etc/sysconfig/ iptables and placed a task with these contents at the location PROD/inputs/tasks/os/ cf.iptables_sync:

```
control:
        any::
                addinstallable          = (      restartiptables )
```

```
copy:
        redhat::
                $(master_etc)/sysconfig/iptables
                        dest=/etc/sysconfig/iptables
                        mode=444
                        owner=root
                        group=root
                        server=$(fileserver)
                        trustkey=true
                        type=checksum
                        encrypt=true
                        define=restartiptables


shellcommands:
        redhat.restartiptables::
                # when config is updated, restart iptables
                "/sbin/service iptables restart"
                        timeout=60 inform=true
```

We activated a new `hostgoup` for Red Hat systems by adding this line to `PROD/inputs/`
`hostgroups/cf.hostgroup_mappings`:

```
redhat::                            hostgroups/cf.redhat
```

Then, we created a `hostgroup` file at `PROD/inputs/hostgroups/cf.redhat` with these
contents:

```
import:
        any::
                tasks/os/cf.iptables_sync
```

It might seem strange to use the `any` class in the `cf.redhat` hostgroup file, but if you
think about it, the task doesn't apply to all hosts on our network, only to the hosts that
import this `hostgroup` file. That means that this `any::` class will actually apply to only Red
Hat systems.

Now, sit back and let NRPE go out to your network. If you encounter any issues while
building NRPE, refer to the `NRPE.pdf` file included in the `docs` directory of the NRPE source
distribution.

Monitoring Remote Systems

So far, we're simply using the example configuration included with Nagios to monitor *only* the system that is actually running Nagios. To make Nagios generally useful, we need to monitor remote systems.

As we progress through Nagios configuration in step 20, the information will come at you very quickly. We recommend that you immediately refer to the documentation on the Nagios web site or one of the recommended Nagios books if anything is unclear. We wish to remind you that Nagios is very flexible, and perhaps because of that, it is rather complicated. There is no substitute for experience, so dig in with us and start becoming familiar with it right away!

Step 20: Configuring Nagios to Monitor All Hosts at Our Example Site

First, we need to create a directory for server host and service objects on the cfengine master and have Nagios look for configuration files in this new directory:

```
# mkdir PROD/repl/root/usr/pkg/nagios-conf/objects/servers/
```

Edit `PROD/repl/root/usr/pkg/nagios-conf/nagios.cfg`, and uncomment this line:

```
#cfg_dir=/usr/pkg/nagios/etc/servers
```

Then, change it so that it looks like this:

```
cfg_dir=/usr/pkg/nagios/etc/objects/servers
```

You should also change the default `admin_email` and `admin_pager` addresses in `nagios.conf` to something appropriate for your site:

```
admin_email=admins@example.org
admin_pager=admins@example.org
```

We then turned on regular expression matching in `nagios.conf` with this line (needed for the regular expressions that we use later in service object definitions):

```
use_regexp_matching=1
```

We copied the `linux-server` template in `PROD/repl/root/usr/pkg/nagios-conf/objects/templates.cfg` to a second similar section to create a new `unix-server` template that is set to issue alarms on a 24 × 7 schedule. Here is the new `unix-server` template definition:

```
define host{
        name            unix-server   ; The name of this host template
        use     generic-host          ; inherits from generic-host template
        check_period    24x7          ; Unix hosts are checked round the clock
        check_interval           5    ; Actively check the host every 5 minutes
        retry_interval           1    ; Schedule host check retries every minute
        max_check_attempts  10 ; Check each Linux host 10 times (max)
        check_command        check-host-alive ; Default command to check Unix hosts
        notification_period      24x7 ; Always alarm
        notification_interval    120  ; Resend notifications every 2 hours
        notification_options     d,u,r ; notify for specific host states
        contact_groups           admins ; Notify admins by default
        register                 0    ; DONT REGISTER- ITS JUST A TEMPLATE!
        }
```

Templates are used in Nagios to avoid repeating the same values for every service and host object. These objects have many required entries, but Nagios allows the use of templates that contain all the required values. We can use the template instead of listing every required value in the objects that we define. Template definitions are very similar to the host or service definitions that they are meant for, but templates contain the line `register 0` to keep Nagios from loading it as a real object. Any or all values can be overridden in an object definition that utilizes a template.

■Note Be aware that `escalation` settings override the `contact_groups` setting in service definitions. We have no `escalation` settings and won't configure it in this chapter, but keep them in mind for your own configurations.

Now that we have a template that suits our needs, we can inherit from it in our service definitions and specify only important values or those that we wish to override from the template's values.

In the directory `PROD/repl/root/usr/pkg/nagios-conf/objects/servers`, we have four files to define the objects to monitor on our network:

- hosts.cfg

- hostgroups.cfg

- system_checks.cfg

- web_checks.cfg

First, we define the hosts at our site in the file hosts.cfg:

```
define host{
        use                     unix-server
        host_name         hemingway.campin.net
        }
define host{
        use                     unix-server
        host_name         goldmaster.campin.net
        }
define host{
        use                     unix-server
        host_name         aurora.campin.net
        }
define host{
        use                     unix-server
        host_name         rhlamp.campin.net
        }
define host{
        use                     unix-server
        host_name         rhmaster.campin.net
        }
define host{
        use                     unix-server
        host_name         loghost1.campin.net
        }
define host{
        use                     unix-server
        host_name         etchlamp.campin.net
        }
```

Nagios host definitions allow the specification of the host's IP address. We purposely leave out that IP address because we want Nagios to use the DNS to find it, for two reasons:

- If we change the host's IP address, we want to only have to change it in the DNS, not in Nagios as well. We might forget and cause confusing alarms.

- We normally rely on the DNS for normal function at our site, so if there are DNS problems, we will allow it to cause failed checks in monitoring as well. We don't want to mask broken DNS in Nagios by avoiding it, we want to always use the DNS and see the problems.

Now that we have host definitions for all the hosts that we want to monitor at our site, we will set up groups in the file `hostgroups.cfg`:

```
define hostgroup{
        hostgroup_name  campin-web-servers    ; The name of the hostgroup
        members              rhlamp.campin.net     ; Comma separated list of hosts
        }
define hostgroup{
        hostgroup_name  infrastructure-web        ; The name of the hostgroup
        members              etchlamp.campin.net    ; Comma separated list of hosts
        }
define hostgroup{
        hostgroup_name  all-servers      ; The name of the hostgroup
        members              .*                      ; Comma separated list of hosts
        }
```

Using `hostgroups` this way allows us to easily add additional systems to Nagios that perform the same functions as existing hosts. We will have to add only the new host to an existing `hostgroup` and immediately have the proper checks performed against it.

Next, we set up some system level monitoring using NRPE, configured in the file `system.cfg`:

```
define service{
        use                       generic-service
        hostgroup_name            all-servers
        service_description       PING
        check_command             check_ping!100.0,20%!500.0,60%
        service_description       Ping check
        }

define service{
        use                            generic-service
        hostgroup_name                 all-servers
        service_description            SSH
        check_command             check_ssh
        service_description            Remote SSH check
        }
```

```
define service{
        use                                 generic-service
        hostgroup_name            all-servers
        check_command             check_nrpe!check_zombie_procs
        service_description        Zombie process check over NRPE
        }

define service{
        use                                 generic-service
        hostgroup_name            all-servers
        check_command             check_nrpe!check_load
        service_description        Load check over NRPE
        }
```

In the `check_command` field of the preceding service definition, the bang character (!) is used to pass arguments to a command. We defined the `check_nrpe` command definition in the `PROD/repl/root/usr/pkg/nagios-conf/commands.cfg` file with this entry:

```
define command{
        command_name    check_nrpe
        command_line       $USER1$/check_nrpe -H $HOSTADDRESS$ -c $ARG1$
        }
```

This entry means that the `check_nrpe` command is passed the argument `check_load` for the `Load check over NRPE` service. When you look at the command definition for `check_nrpe`, you can now see that what is run on the monitoring host is:

```
# /usr/pkg/nagios-plugins/libexec/check_nrpe -H rhlamp -c check_load
```

Being able to understand and test what Nagios is actually running, as we worked out previously, will be useful in the future when a remote NRPE check malfunctions. Monitoring systems are complicated, and a failure might happen in the monitoring system itself. Being able to manually test the commands that Nagios runs will prove useful.

Next, we set up some web server checks in the file `web_checks.cfg`:

```
define service{
        use                                  generic-service
        hostgroup_name        infrastructure-web
        service_description     HTTPS
        check_command         check_https!443!/
        }
```

```
define service{
        use                          generic-service
        hostgroup_name       campin-web-servers
        service_description    HTTP check
        check_command        check_http
        }
```

We defined the `check_https` check earlier to test the web server on localhost, so here we simply set it up for a remote host and it works properly.

Each time we update the Nagios configuration files, cfengine gets the files to the correct location on our monitoring host (*etchlamp*) and restarts the Nagios daemon.

We can rest easy knowing that if the *etchlamp* system fails due to hardware issues, we will simply need to reimage the host, and without any manual intervention cfengine will set the host up for us again. What a great feeling!

Step 21: Party!

That was a lot of work, but now that it's complete, we think that some celebration is appropriate. Let's look at what we've accomplished.

We've deployed a very complex software framework across an environment with three different platforms in an automated manner. We are enjoying the full benefits of automation:

- Easy updates to any monitoring component's configuration files

- Easy program file updates for Nagios, NRPE, or the Nagios plug-ins

- Easy restoration to full functionality if any hosts suffer full system failure, even the central monitoring host

At this point, we have the four components of Nagios deployed, as planned: Nagios itself, the Nagios plug-ins, the Nagios web interface, and NRPE. We can extend the system to run plug-ins that we define, either locally on systems via NRPE or across the network to test client/server applications.

We shouldn't need to change anything about the framework in the near future, only add checks and perhaps new plug-ins. Our monitoring infrastructure choice really shines in the easy addition of new plug-ins; it should be able to support us for quite a while without any core modifications.

What Nagios Alerts Really Mean

When notifications (i.e., alerts) are sent from Nagios, or for that matter from any monitoring system, what does it really mean?

The immediate answer from most SAs is similar to "it means a host or service failed." This isn't really true. The truth of the matter is that *a monitoring program or script signaled failure*. When Nagios sends a notification, it means that a plug-in script exited with an exit code that was something other than exit code zero (exit code zero means "okay" to Nagios).

If the plug-in is `check_http`, you might assume that it means that a remote web server is down, but what if a static file at the URL that `check_http` is requesting was moved? Will a 404 HTTP status (which means "document not found") cause `check_http` to fail? Do you even know the answer to that? If not, you should find out. What if the monitoring host has a bad route entry that causes traffic to the web server to timeout but doesn't stop notifications from reaching you? The web server itself is probably fine and is probably reachable by all systems *except* the monitoring host.

Don't jump to the conclusion that a notification means that a service or host has failed. You need to understand exactly what each service definition is checking and validate that the service is really failing with some checks of your own before undertaking any remediation steps.

Ganglia

Ganglia is a distributed monitoring system that uses graphs to display the data it collects. Nagios will let us know if an application or host is failing a check, but Ganglia is there to show us long-term trends in host resource utilization and performance. You can also feed site-specific metrics into Ganglia, though we don't demonstrate doing so in this book.

If a host intermittently triggers a load alarm in Nagios, with no clear cause immediately visible, looking at graphs of the system's load over time can be useful in helping you see when the load increase began. Armed with this information, we can check if the alarm correlates to a system change or application update. Ganglia is extremely useful in such situations, as it generates graphs showing important host metrics such as CPU utilization, system load, and disk and network utilization.

Ganglia is also useful to visualize trends in resource usage with an eye toward capacity planning. If you observe a steady rise in CPU or memory utilization on your web server, you can use this information to justify hardware upgrades or the purchase of more systems to share the load.

We could use many other open source software packages for host resource graphing (and we have in the past). Some of them are more general purpose than Ganglia and

some even plug directly into Nagios. We like Ganglia because it is fast and efficient, scales incredibly well, and adding new custom metrics to the Ganglia graphs is extremely easy.

The core functionality of Ganglia is provided by two main daemons, along with a web front end:

- `gmond`: This multithreaded daemon runs on each host you want to monitor. `gmond` keeps track of state on the system, relays the state changes on to other systems via TCP or multicast UDP, listens for and gathers the state of other `gmond` daemons in the local cluster, and answers request for all the collected information. The `gmond` configuration will cause hosts to join a cluster group. A site might contain many different clusters, depending on how the administrator wants to group systems for display in the Ganglia web interface.

- `gmetad`: This daemon is used to aggregate Ganglia data and can even be used to aggregate information from multiple Ganglia clusters. `gmetad` polls one or many `gmond` daemons or other `gmetad` daemons, parses the collected XML, stores the information in RRD files (round-robin databases), and exports the XML over TCP sockets to clients.

- *Web interface*: Written in PHP, it connects to a local `gmetad` daemon to receive the XML tree needed to display the Ganglia data. Information can be viewed sitewide, clusterwide, or for a single host over periods of time such as the last hour, day, week, or month. The web interface uses graphs generated by `gmetad` to display historical information.

Ganglia's `gmond` daemon can communicate using TCP with explicit connections to other hosts that aggregate a cluster's state, or it can use multicast UDP to broadcast the cluster state to all listening hosts. We go with TCP and explicitly name aggregator hosts and then poll those hosts explicitly with `gmetad`. The `gmond` configuration file still has UDP port configuration settings, but they won't be used at our example site.

Building and Distributing the Ganglia Programs

Ganglia needs to be compiled for each platform at our site. We built Ganglia on Solaris, Red Hat, and Debian Linux by downloading and installing with the following sequence of commands. Note that a C++ compiler will need to be present on the system, as well as development libraries for RRDtool (and the package `libpng12-0` on Debian). Without the RRDtool libraries the build will seem successful, but the `gmetad` program will fail to be built.

```
# wget http://internap.dl.sourceforge.net/sourceforge/ganglia/ganglia-3.0.7.tar.gz
# gunzip -dc ganglia-3.0.7.tar.gz | tar xf -
# cd ganglia-3.0.7
# ./configure --prefix=/usr/pkg/ganglia-3.0.7 && make
# sudo make install
# rsync -avze ssh /usr/pkg/ganglia-3.0.7/ \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/usr/pkg/➥
ganglia-3.0.7-i686.debian
```

As shown in the preceding set of commands, we copied the resulting /usr/pkg/
ganglia-3.0.7 binaries from each platform to the appropriate directory in the master
files tree on the cfengine master (though the preceding command only demonstrates the
Debian i686 build). Here are the three directories:

```
PROD/repl/root/usr/pkg/ganglia-3.0.7-i686.debian/
PROD/repl/root/usr/pkg/ganglia-3.0.7-i686.redhat/
PROD/repl/root/usr/pkg/ganglia-3.0.7.sunos_sun4u/
```

The gmond binary will use a built-in configuration if it can't find its default configu-
ration file at /etc/gmond.conf (or it isn't started with the command line option -c to
manually specify a configuration file). To see the default configuration run gmond with this
argument:

```
#  gmond --default_config > gmond.conf
```

You can then redirect the output to a file (named gmond.conf), edit as appropriate for
your site, and then place the gmond.conf file on the cfengine master. The beautiful thing
about this option is that it even emits comments describing each configuration section!
Ganglia was clearly written by system administrators.

We did precisely this to get started on our configuration and then changed the file to
suit our needs. Here are the portions of gmond.conf that we changed:

```
 globals {
  setuid = no
  user = daemon
  user = nobody
  host_dmax = 3600
  cleanup_threshold = 300 /*secs */
}

cluster {
  name = "Campin.net"
}
```

```
udp_send_channel {
  host = goldmaster
  port = 8649
}

udp_send_channel {
  host = etchlamp
  port = 8649
}

udp_recv_channel {
  mcast_join = 239.2.11.71
  port = 8649
}

udp_recv_channel {
  port = 8649
}

tcp_accept_channel {
    acl {
            default = "deny"
            access {
                    ip=127.0.0.1
                    mask=32
                    action = "allow"
            }
            access {
                    ip=192.168.1.239
                    mask=32
                    action = "allow"
            }
    }
  port = 8649
}
```

We kept the default Ganglia port of 8649 (which spells "UNIX" on a T9 phone keypad). We set the hosts *goldmaster* and *etchlamp* to be the cluster data aggregators via the udp_send_channel sections. We'll use gmetad to poll the cluster state from these two hosts. The tcp_accept_channel section allows our host running gmetad (192.168.1.239 for *etchlamp*) to poll state over TCP from any host running gmond. The rest of the configuration file is unchanged.

We got started with the example gmetad.conf file from the Ganglia source distribution at the location gmetad/gmetad.conf. We placed the Ganglia configuration files (gmond.conf and gmetad.conf) into the directory PROD/repl/root/usr/pkg/ganglia-conf on the cfengine master. We'll modify the contents of the example gmetad.conf later in the chapter.

We added a UNIX/Linux user account called ganglia to the PROD/repl/root/etc/ [passwd|shadow|group] files with these entries:

- /etc/passwd: ganglia:x:106:109:Ganglia Monitor:/usr/pkg/ganglia:/bin/false

- /etc/group: ganglia:x:109:

- /etc/shadow: ganglia:!:14103:0:99999:7:::

Next, we created a cfengine task for copying out the binaries at the location PROD/ inputs/tasks/app/ganglia/cf.sync_ganglia_binaries on the cfengine master:

```
classes:  # synonym groups:
        have_usr_pkg_ganglia_3_0_7      = ( IsDir(ganglia-3.0.7-i686.debian) )

control:
        any::
            AddInstallable       = ( ganglia_binaries_updated ganglia_conf_updated )
            AllowRedefinitionOf    = ( ganglia_master_dir )
            dest_dir                        = ( "ganglia-3.0.7" )

        debian_4_0.i686::
                ganglia_master_dir      = ( "ganglia-3.0.7-i686.debian" )

        redhat.i686::
                ganglia_master_dir      = ( "ganglia-3.0.7-i686.redhat" )

        solaris|solarisx86::
                ganglia_master_dir      = ( "ganglia-3.0.7.sunos_sun4u" )

copy:
        any::
                $(master)/repl/root/usr/pkg/$(ganglia_master_dir)
                              dest=/usr/pkg/$(dest_dir)
                              mode=755
                              r=inf
                              owner=root
                              group=root
                              type=checksum
```

```
                              server=$(fileserver)
                              encrypt=true
                              define=ganglia_binaries_updated

              $(master)/repl/root/usr/pkg/ganglia-conf
                              dest=/usr/pkg/ganglia-conf
                              mode=755
                              r=inf
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
                              define=ganglia_conf_updated


shellcommands:
      ganglia_binaries_updated::
              # All hosts run gmond. Restart it completely when binaries update
              "/usr/bin/pkill gmond ; sleep 1 ; /usr/bin/pkill -9 gmond ; ➥
/usr/pkg/ganglia/sbin/gmond -c /usr/pkg/ganglia-conf/gmond.conf "
                      timeout=60 inform=true owner=daemon


      ganglia_web.ganglia_binaries_updated::
              # the box running the ganglia web interface runs gmetad, restart it
              #  when the binaries update
              "/usr/bin/pkill gmetad ; sleep 1 ; /usr/bin/pkill -9 gmetad ; ➥
/usr/pkg/ganglia/sbin/gmetad -c /usr/pkg/ganglia-conf/gmetad.conf "
                      timeout=60 inform=true owner=daemon


processes:
      any::
              "gmond" restart
                "/usr/pkg/ganglia/sbin/gmond -c /usr/pkg/ganglia-conf/gmond.conf"
                inform=true umask=022 owner=daemon


      ganglia_conf_updated::
              "gmond" signal=hup inform=true
              "gmetad" signal=hup inform=true


links:
      any::
              /usr/pkg/ganglia ->! /usr/pkg/$(dest_dir)
```

Next, add this line to `PROD/inputs/hostgroups/cf.any` so that all of our hosts get the Ganglia programs copied over:

```
tasks/app/ganglia/cf.sync_ganglia_binaries
```

Note that we don't place a start-up script onto the systems for Ganglia. We simply have cfengine start the appropriate daemons if they aren't found in the system's process list. This places an obvious dependency on having `cfexecd` running, calling `cfagent` regularly. We always start up cfengine at boot on all systems at our site, so this shouldn't be a problem.

Configuring the Ganglia Web Interface

Our central Ganglia machine will run the web interface for displaying graphs, as well as the `gmetad` program that collects the information from the `gmond` daemons on our network.

Ganglia's web interface is written in PHP and distributed in the source package. Copy the PHP files from the Ganglia source package's web directory to this location on the cfengine master:

```
# tar zxf ganglia-3.0.7.tar.gz
# cd ganglia-3.0.7
# mkdir -p /var/lib/cfengine2/masterfiles/PROD/repl/root/var/www/apache2-default
# cp -r web \
/var/lib/cfengine2/masterfiles/PROD/repl/root/var/www/apache2-default/ganglia
```

We will use cfengine to copy this directory to our host named *etchlamp*, which already has a web server will serve as our network's Ganglia console. Again, we used the directory `PROD/inputs/tasks/app/ganglia` on the cfengine master and put the task `cf.setup_ganglia_web` in it with these contents:

```
control:
        ganglia_web.debian::
                addinstallable          = (     restart_apache2 )

copy:
        ganglia_web.debian::
                $(master)/repl/root/var/www/apache2-default/ganglia
                                dest=/var/www/apache2-default/ganglia
                                mode=555
                                r=inf
                                purge=false
                                owner=root
```

```
                                    group=root
                                    type=checksum
                                    server=$(fileserver)
                                    encrypt=true
                                    define=restart_apache2

                $(master_etc)/apache2/sites-available/ganglia
                                    dest=/etc/apache2/sites-available/ganglia
                                    mode=444
                                    owner=root
                                    group=root
                                    type=checksum
                                    server=$(fileserver)
                                    encrypt=true
                                    define=restart_apache2

                $(master_etc)/apache2/ssl/ganglia.pem
                                    dest=/etc/apache2/ssl/ganglia.pem
                                    mode=444
                                    owner=root
                                    group=root
                                    type=checksum
                                    server=$(fileserver)
                                    encrypt=true
                                    define=restart_apache2


directories:
        ganglia_web.debian::
                /var/www/apache2-default/ganglia  mode=755 owner=root
                                            group=root inform=true
                /usr/pkg/ganglia-data/rrds      mode=755 owner=daemon
                                        group=daemon inform=true


processes:
        ganglia_web.debian::
                "/usr/sbin/apache2" restart "/etc/init.d/apache2 start"
                                    inform=true umask=022

                "gmetad" restart
                "/usr/pkg/ganglia/sbin/gmetad -c /usr/pkg/ganglia-conf/gmetad.conf "
                    inform=true umask=022 owner=daemon
```

```
shellcommands:
        ganglia_web.debian.restart_apache2::
                "/etc/init.d/apache2 restart"
                        timeout=60
                        umask=022

links:
        ganglia_web.debian::
                /etc/apache2/sites-enabled/ganglia ->!
                        /etc/apache2/sites-available/ganglia

                # the make-ssl-cert utility created this link when we created
                # ganglia.pem, we'll preserve it using cfengine
                /etc/apache2/ssl/4c1b6a93          ->! /etc/apache2/ssl/ganglia.pem
```

This task causes the `gmetad` daemon to be started on the `ganglia_web` host if it isn't running (we define `ganglia_web` in the next section). Our configuration for the `gmetad` daemon (`PROD/repl/root/usr/pkg/ganglia-conf/gmetad.conf`) follows:

```
data_source "Campin.net" 60 goldmaster etchlamp 8649
gridname "Campin"
all_trusted on
setuid off
rrd_rootdir "/usr/pkg/ganglia-data/rrds"
```

We removed all comments to make the file easy to read. The comments in the example configuration in the Ganglia source directory (`gmetad/gmetad.conf`) are extensive and serve as sufficient documentation to get most users going with a working configuration.

Next, we needed to generate the ganglia SSL certificate for our Ganglia web site and put it on the cfengine master:

```
# /usr/sbin/make-ssl-cert /usr/share/ssl-cert/ssleay.cnf \
/etc/apache2/ssl/ganglia.pem
# scp /etc/apache2/ssl/ganglia.pem \
goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/apache2/ssl/
```

To configure the `ganglia_web` role in cfengine, we added this line to `PROD/inputs/classes/cf.main_classes`:

```
ganglia_web             = ( etchlamp )
```

Our Debian-based Ganglia web system needs some additional packages. To install them at initial system installation time, we added the packages `rrdtool` and `libpng12-0`

into the FAI package list for the `WEB` class. We installed them manually using `apt-get` on *etchlamp* in this case, so that we didn't have to reimage the host just to add two packages.

Next, we created a new `hostgroup` file for our new `ganglia_web` role on the cfengine master at the location `PROD/inputs/hostgroups/cf.ganglia_web`, with these contents:

```
import:
        any::
                tasks/app/apache/cf.setup_ganglia_web
```

Then, we added this to `PROD/inputs/hostgroups/cf.hostgroup_mappings`:

```
ganglia_web::                    hostgroups/cf.ganglia_web
```

Once cfengine on *etchlamp* copies the PHP content and Apache configuration files, we can visit `https://ganglia.campin.net/` in our web browser and view graphs for all the hosts at our site, individually or as a whole. If you haven't previously used a similar host resource graphing system as part of your monitoring suite, you'll be amazed at how often you refer to the graphs during troubleshooting or for capacity planning.

Now You Can Rest Easy

At this point, we have a full monitoring suite at our site with Ganglia and Nagios. We can utilize Nagios for 24×7 alerting on host and service availability, and we can utilize Ganglia to view short and long-term system resource usage. Both are extremely flexible and will grow and scale along with our new infrastructure.

As your site requires more and more monitoring, you might benefit from the distributed monitoring capabilities of Nagios. Nagios version 3.0 and above has a much improved ability to operate in such a fashion. Utilizing cfengine, you can easily deploy a test instance of distributed Nagios in order to determine if the additional load sharing and redundancy is a good fit for your site. Many sites simply purchase more powerful hardware in order to utilize Nagios against many hosts and services, but at some point, this may no longer be feasible.

Ganglia will scale extremely well to large numbers of systems, and most of the follow-on configuration will be around breaking up hosts into separate groups, and possibly utilizing multicast. If you don't use multicast, you'll want to utilize many `gmond` instances to aggregate the cluster's state and simply configure `gmetad` to poll the cluster state from a list of several hosts running `gmond`. This allows one or more `gmond` aggregators to fail and still have Ganglia function properly. We only use two at our example site, you may choose to run with many more as the total number of systems at your site increases.

Infrastructure Enhancement

At this point, we have a fully functional infrastructure. We have automated all of the changes to the hosts at our site from the point at which the initial imaging hosts and cfengine server were set up.

We're running a rather large risk, however, because if we make errors in our cfengine configuration files, we won't have an easy way to revert the changes. We run an even greater risk if our cfengine server were to suffer hardware failure: we would have no way of restoring the cfengine `masterfiles` tree. The other hosts on our network will continue running cfengine, and they will apply the last copied policies and configuration files, but no updates will be possible until we restore our central host.

Subversion can help us out with both issues. Using version control, we can easily track the changes to all the files hosted in our cfengine `masterfiles` tree, and by making backups of the Subversion repository, we can restore our cfengine server in the event of system failure or even total site failure.

Cfengine Version Control with Subversion

With only a small network in place, we already have over 2,800 lines of configuration code in over 55 files under the `PROD/inputs` directory. We need to start tracking the different versions of those files as time goes on, as well as tracking any additional files that are added. The workplace of one of this book's authors has over 30,000 lines of cfengine configuration in 971 files. Without version control, it is difficult to maintain any semblance of control over your cfengine configuration files, as well as the files being copied by cfengine.

We covered basic Subversion usage in Chapter 8 and included instructions on how to set up a Subversion server with an Apache front end. We'll utilize that infrastructure to host version control for our cfengine master repository.

Importing the masterfiles Directory Tree

In order to import our cfengine `masterfiles` directory into Subversion, we need to create the repository on *etchlamp*, our Subversion host. Conveniently, we already created the

repository with cfengine back in Chapter 8 and granted read/write access to the `nate` and `kirk` users.

Now, we want to set up a read-only user to be used to check out changes to production hosts. Once we check out a copy of the production cfengine `masterfiles` tree, we don't want to allow changes to be checked in directly from that tree. We want our administrators to edit a working copy of the configuration, check in their changes, and then have the production working copy updated directly from Subversion.

To set up the read-only user, create it manually on the system *etchlamp* (as the root user), and copy the access file to the cfengine master:

```
# htpasswd /etc/apache2/dav_svn.passwd readonly
New password:
Re-type new password:
Adding password for user readonly
# scp /etc/apache2/dav_svn.passwd \
 goldmaster:/var/lib/cfengine2/masterfiles/PROD/repl/root/etc/apache2/dav_svn.passwd
```

Now, we want to grant read-only access to the cfengine Subversion repository to this new user. Change this section in `PROD/repl/root/etc/apache2/svn_accessfile`

```
[cfengine:/]
@admins = rw
```

to this

```
[cfengine:/]
@admins = rw
readonly = r
```

Before we import into the Subversion repository, we'll want to make sure that all the `.svn` directories that get added into the `masterfiles` tree don't get copied out to clients later on. These are unnecessary and are a bit of a security risk. We'll accomplish this with a global `ignore` action. Create the directory `PROD/inputs/ignore` on the cfengine master, and place these contents in a new file at `PROD/inputs/ignore/cf.ignore`:

```
ignore:
       any::
               .svn
```

Import this file into `cfagent.conf`. Since the file is made up entirely of imports, you can place this entry anywhere after the `import:` line:

```
       # globally ignore certain files and directories
       ignore/cf.ignore
```

At this point we're ready to import the masterfiles/PROD directory. On the cfengine master *goldmaster*, run these commands:

```
# svn --username=nate import /var/lib/cfengine2/masterfiles/PROD \
https://svn.campin.net/svn/cfengine/masterfiles/PROD \
-m"initial cfengine import of the PROD tree only"
Authentication realm: <https://svn.campin.net:443> Campin.net Subversion Repository
Password for 'nate':
Adding      /var/lib/cfengine2/masterfiles/PROD/repl
Adding      /var/lib/cfengine2/masterfiles/PROD/repl/root
Adding      /var/lib/cfengine2/masterfiles/PROD/repl/root/var
Adding      /var/lib/cfengine2/masterfiles/PROD/repl/root/var/www
Adding      /var/lib/cfengine2/masterfiles/PROD/repl/root/var/www/html
```

The output went on for some time; it's quite surprising just how many files we have in there at this point. The large number of files highlights the importance of keeping our files in Subversion, if only as a backup measure. The utility of version control for our repository goes far beyond simple backups, as you will see in the next section.

Now, when you visit the URL https://svn.campin.net/svn/cfengine/masterfiles/PROD/ in a web browser, you'll see your repl and inputs directories in Subversion, with revision 1. To use our current masterfiles/PROD tree from Subversion on our cfengine master, we'll need to check out a working copy in place of the current PROD directory. Here are the commands we ran (as the root user) on *goldmaster*:

```
# cd /var/lib/cfengine2/masterfiles
# mv PROD PROD.bak
# svn --username=readonly co https://svn.campin.net/svn/cfengine/masterfiles/PROD
A    PROD/inputs
A    PROD/inputs/cfservd.conf
A    PROD/inputs/control
A    PROD/inputs/control/cf.friendstatus
A    PROD/inputs/control/cf.control_cfagent_conf
A    PROD/inputs/control/cf.control_cfexecd
A    PROD/inputs/cf.preconf
```

The output went on for quite some time as all the files were checked out. In order to edit those files, we can (and will) check out the tree somewhere else, such as in our home directory. This way, we will be working on changes in an environment where they won't immediately be copied by the systems on our network.

■ Note We should never again work directly on the files in the PROD tree on the cfengine master and check in our changes from there. It is bad practice to directly edit the live files used for configuration at our site.

If you attempt to check in files from the `/var/lib/cfengine2/masterfiles/PROD` tree on the cfengine master, you'll get errors like this:

```
# cd PROD
# touch foo
# svn add foo
# svn commit -m"this shouldn't work"
svn: Commit failed (details follow):
svn: MKACTIVITY of '/svn/cfengine/!svn/act/7900b02f-eb97-4954-8ea6-8645e662404e':
403 Forbidden (https://svn.campin.net)
```

We got this error because we checked out the tree as the `readonly` user, and that user lacks the privileges to check back in. We could of course specify the `--username=nate` argument to the Subversion client, which would allow the check-in, but that's bad practice. We want to carefully test our changes and have clients see our modifications only once we've committed them. We also ensure that all files are properly checked into version control this way. We don't want administrators to copy files manually into the `PROD` tree, because if the cfengine server fails and we restore from Subversion, we would be missing some of our configuration files! We need to avoid this at all costs. Always update the `/var/lib/cfengine2/masterfiles/PROD` tree only with Subversion updates.

Another risk from working directly on the live `PROD` tree is we might accidentally save a working copy of a file such as `cfagent.conf` without meaning to. If we're working on an offline working copy, we don't have to worry about such accidents. Start developing good habits now.

WORKING WITH A CFENGINE MASTERFILES WORKING COPY

If you check out a working copy of the `masterfiles/PROD` tree to develop changes, your modifications won't be seen by the cfengine clients on your network immediately after you check in your changes. To make the changes visible, you'll need to check out the changes into the `/var/lib/cfengine2/masterfiles/PROD` directory on your cfengine master.

We've created the `readonly` user so that a nonprivileged Subversion account can be shared for this checkout task. We don't ever want to use a real person's Subversion account for such a checkout, since any other staff with root privileges could check in changes as that user, impersonating them.

You could certainly set up an automated check out of the latest version of the `PROD` tree onto to the cfengine master, but that's probably not a good idea at this time. We'll want to check out changes manually, so that we can be sure that we really want the changes contained in those updates.

We would like to know when changes are checked into the repository so that we see when other administrators make changes that might affect work we're doing or catch errors in their changes. Subversion has a feature called hooks that allows scripts to run when different repository actions happen. You can see the actions for which hooks are supported by inspecting the template hook scripts that the `svnadmin create` command placed in the `hooks` subdirectory in the cfengine repository:

```
# ls /var/svn/repository/cfengine/hooks/
post-commit.tmpl  post-revprop-change.tmpl  pre-commit.tmpl  pre-revprop-change.tmpl
start-commit.tmpl  post-lock.tmpl  post-unlock.tmpl  pre-lock.tmpl pre-unlock.tmpl
```

Inspect the hook template files themselves to see what actions are supported.

To get e-mail notifications when a change is committed to our cfengine Subversion repository, we'll place a shell script at the location `PROD/repl/root/var/svn/repository/cfengine/hooks/post-commit`.

But wait! We can't do this directly on the cfengine master any longer. We'll need to check out our own personal working copy of the cfengine repository. We logged into the system *goldmaster* as our own user account and checked out a working copy with these commands:

```
$ cd ~/
$ svn co https://svn.campin.net/svn/cfengine/masterfiles
```

These commands will give us a working copy at `~/masterfiles`. Since our home directory is shared via NFS, we can work on our working copy of the cfengine `masterfiles` tree from any host on the network that we choose. All of our systems have a Subversion client, so it's a matter of personal preference.

It turns out to have been highly useful that so far in this book we have referenced the path to files in the cfengine `masterfiles` tree as relative to the `PROD` directory, because from now on, the files we're working with will be a working copy, and the base directory will be different for every user. We'll continue referring to files as relative to the `PROD` directory. When we start working with the `DEV` and `STAGE` directory trees, we'll also refer to files and directories stored within as relative to those directories.

Now, we need to create the file `PROD/repl/root/var/svn/repository/cfengine/hooks/post-commit` in our working copy of the cfengine tree. Create the directories with these commands:

```
$ cd ~/masterfiles/PROD
$ mkdir -p repl/root/var/svn/repository/cfengine/hooks
```

Then, create the file `PROD/repl/root/var/svn/repository/cfengine/hooks/post-commit` with these contents:

```
#!/bin/sh
REPOS="$1"
REV="$2"
LOG=`/usr/bin/svnlook log -r $REV $REPOS`
AUTHOR=`/usr/bin/svnlook author -r $REV $REPOS`

/usr/bin/svnnotify --repos-path "$REPOS" --revision "$REV" --with-diff \
--to ops@example.org --from "$AUTHOR" --reply-to ops@example.org --subject-prefix \
 "[CFENGINE SVN]" --subject-cx --no-first-line
```

To have the svnnotify program on our Debian-based Subversion server, we'll
need to install the package libsvn-notify-perl. We added it to the file /srv/fai/config/
package_config/WEB on *goldmaster* (our FAI installation host) back in Chapter 8, so it is
already installed.

Next, we needed to check our new hook script into Subversion. The way we handled
this is to add the highest level directory in the tree that isn't yet checked in:

```
$ svn add repl/root/var/svn
A         repl/root/var/svn
A         repl/root/var/svn/repository
A         repl/root/var/svn/repository/cfengine
A         repl/root/var/svn/repository/cfengine/hooks
A         repl/root/var/svn/repository/cfengine/hooks/post-commit
$ svn commit
```

When you type svn commit without the -m argument (which automatically submits
the log entry), you're dropped into an editor that allows you to enter a comment for the
commit, along with the files (and directories, if applicable) that are being modified in the
current commit. The editor screen for the preceding commit looked like this:

```
--This line, and those below, will be ignored--

A    PROD/repl/root/var/svn
A    PROD/repl/root/var/svn/repository
A    PROD/repl/root/var/svn/repository/cfengine
A    PROD/repl/root/var/svn/repository/cfengine/hooks
A    PROD/repl/root/var/svn/repository/cfengine/hooks/post-commit
```

The cursor was at the top of the screen, where the comments belong. We could see
that all of our new directories were being committed, along with our new file. We entered
a comment about how this is to enable notifications for cfengine repository commits,
saved the file, and saw this Subversion client output:

```
"svn-commit.tmp" 8L, 346C written
Adding          PROD/repl/root/var/svn
Adding          PROD/repl/root/var/svn/repository
Adding          PROD/repl/root/var/svn/repository/cfengine
Adding          PROD/repl/root/var/svn/repository/cfengine/hooks
Adding          PROD/repl/root/var/svn/repository/cfengine/hooks/post-commit
Transmitting file data .
Committed revision 4.
```

Now, we need to set up a task to copy out our new hook script. We'll set it up as a recursive copy of a `hooks` directory, even though we currently have only one hook script. This will allow us to develop other hook scripts later and simply place them into a directory in the `masterfiles` tree to have the new hook script automatically copied to the Subversion server by cfengine.

We created a task at `PROD/inputs/tasks/app/svn/cf.copy_hooks` with these contents:

```
copy:

        svn_server.debian::

                $(master)/repl/root/var/svn/repository/cfengine/hooks

                                dest=/var/svn/repository/cfengine/hooks
                                mode=555
                                r=inf
                                purge=false
                                owner=www-data
                                group=www-data
                                type=checksum
                                server=$(fileserver)
                                encrypt=true
```

We then added this new task to the repository as follows:

```
$ svn add inputs/tasks/app/svn/cf.copy_hooks
A        inputs/tasks/app/svn/cf.copy_hooks
$ svn commit -m"added task to copy out the cfengine svn repo hooks directory"
Adding          PROD/inputs/tasks/app/svn/cf.copy_hooks
Transmitting file data .
Committed revision 5.
```

We still need to activate this task by importing it, so we added this line to `PROD/inputs/hostgroups/cf.svn_server`:

```
tasks/app/svn/cf.copy_hooks
```

We issued the `svn commit` command again, and now, our Subversion repository should have all the changes required to copy out our new hook to our Subversion repository. We still need to update our live `masterfiles/PROD` working copy on the cfengine master. As root on *goldmaster*, we issued these commands:

```
# cd /var/lib/cfengine2/masterfiles/PROD
# svn update
A    inputs/hostgroups/cf.svn_server
A    inputs/tasks/app/svn/cf.copy_hooks
A    repl/root/var/svn
A    repl/root/var/svn/repository
A    repl/root/var/svn/repository/cfengine
A    repl/root/var/svn/repository/cfengine/hooks
A    repl/root/var/svn/repository/cfengine/hooks/post-commit
Updated to revision 6.
```

Now, we just need to wait for cfengine to run again on *etchlamp* (the Subversion server) so that it gets the new hook script. After the next `cfagent` run (it runs every 20 minutes at our site), we committed a new version of `PROD/inputs/hostgroups/cf.svn_server` with a blank line added to the end, just to test the notifications. We got this e-mail shortly thereafter:

```
From:  ops@example.org
To:      ops@example.org
Date:   Mon, Sep 1, 2008 at 2:21 AM
Subject: [CFENGINE SVN] [7] masterfiles/PROD/inputs/hostgroups/cf.svn_server

Revision: 7
Author:   nate
Date:     2008-09-01 02:21:26 -0700 (Mon, 01 Sep 2008)

Log Message:
-----------
just a blank line to test email notifications

Modified Paths:
--------------
   masterfiles/PROD/inputs/hostgroups/cf.svn_server
```

```
Modified: masterfiles/PROD/inputs/hostgroups/cf.svn_server
===================================================================
--- masterfiles/PROD/inputs/hostgroups/cf.svn_server 2008-09-01 09:16:24 UTC (rev 6)
+++ masterfiles/PROD/inputs/hostgroups/cf.svn_server 2008-09-01 09:21:26 UTC (rev 7)
@@ -8,3 +8,4 @@
                tasks/app/svn/cf.setup_svn_plus_apache
                tasks/app/svn/cf.copy_hooks

+
```

The output displays our new blank line with a simple plus sign, followed by nothing (nothing but a `newline` character, of course).

You can see how useful these e-mail notifications will be when multiple people are committing to the repository. It can also be used for peer review of changes. Standard practice at your site could be to have a meeting where all commits are peer reviewed before the working production copy is updated with the changes committed to the repository.

The major problem with such a system is that there is no mechanism set up to test the changes before they are pushed to the live environment. A typographical error can easily be missed during peer review, causing cfengine to fail to execute properly on all hosts at our site. Clearly a better mechanism is needed. In the next section, we'll explore a way to try out our changes in a nonproduction environment.

Using Subversion to Implement a Testing Environment

We initially set up our cfengine clients to use files under the PROD directory. In this section, we'll start to make use of the DEV directory, which is at the same level as PROD in the masterfiles tree.

To create a new branch in the repository, simply use the `svn copy` command with two URL paths in the repository. First, we made sure the repository has the required base paths; then, we created the branch:

```
$ cd ~/masterfiles/
$ mkdir -p DEV/branches
$ svn add DEV
$ svn commit -m"creating DEV/branches directory structure"
Adding         DEV
Adding         DEV/branches
```

```
Committed revision 8.
$ svn copy https://svn.campin.net/svn/cfengine/masterfiles/PROD \
https://svn.campin.net/svn/cfengine/masterfiles/DEV/branches/1 \
-m"creating the first cfengine development branch"

Committed revision 9.
```

Now, we have a branch for development at `DEV/branches/1` inside the repository. In order to work with it, we'll need to check it out:

```
$ cd ~/masterfiles/DEV/branches/
$ svn co https://svn.campin.net/svn/cfengine/masterfiles/DEV/branches/1
A    1/inputs
A    1/inputs/cfservd.conf
A    1/inputs/control
A    1/inputs/control/cf.friendstatus
A    1/inputs/control/cf.control_cfagent_conf
A    1/inputs/control/cf.control_cfexecd
A    1/inputs/cf.preconf
A    1/inputs/ignore
A    1/inputs/ignore/cf.ignore
...output truncated...
```

Note that inside the repository the branches don't take up much extra space. Subversion has a cheap copy mechanism where branches are really more like hard links to the original copy. The branch really only starts taking up space as it is modified and added to. Be aware that our checkout of the branch does take up the full amount of space in our local filesystem.

Creating arbitrarily named branches in the version repository under `DEV` is fine. We'll be able to check out multiple trees under `DEV` on the cfengine master and point clients at any tree of our choosing. Let's set up that branch now. On the cfengine master host (as the root user), check out the new development branch to the live tree where cfengine clients pull files:

```
# pwd
/var/lib/cfengine2/masterfiles/PROD
# cd ../DEV/
# mkdir branches
# cd branches/
# svn --username=readonly co \
https://svn.campin.net/svn/cfengine/masterfiles/DEV/branches/1
```

Now that we have a development tree available on the cfengine master, we need a nonproduction host to use it on. We don't have any hosts that aren't important to our network, or more importantly to our business, so we'll image a new one. We'll call it *ops1*, meaning that it belongs to the operations team, and use it for testing. We'll create a Debian i686 host, since that's what we use for most of our system roles at this point.

Here are the summarized steps to create the new Debian host:

1. Add entries for the new host to the DNS. We created a forward entry in the file `db.campin.net` and a reverse entry in the file `db.192.168`. As is now the norm, we had to commit the changes to Subversion and update the Subversion working copy on the cfengine master.

2. We'll set up FAI on *goldmaster* to image the host, which means adding an entry to boot the new host in `/etc/dhcp3/dhcpd.conf` and running the command `fai-chboot -IFv ops1`.

3. Image the new host. We don't need to do anything custom to it at this point, so we didn't add it to any special classes in FAI. We want it to be a very basic system.

Now, we needed to change some core files in cfengine in order to have *ops1* utilize the `DEV` tree. In `PROD/inputs/update.conf`, we added these lines to the top:

```
classes:
        dev_servers             = (      ops1
                                         )
```

Then, we added these lines to the `control` section in `PROD/inputs/update.conf`:

```
    any::
            AllowRedefinitionOf  = ( branch )
            branch           = ( PROD )

    dev_servers::
            branch           = ( "DEV/branches/1" )

    any::
            master_cfinput = ( /var/lib/cfengine2/masterfiles/$(branch)/inputs )
```

and we removed this line from `PROD/inputs/update.conf`:

```
master_cfinput = ( /var/lib/cfengine2/masterfiles/PROD/inputs )
```

In `PROD/control/cf.control_cfagent_conf`, the section that looked like this

```
        branch          = ( PROD )
        master_cfinput = ( /var/lib/cfengine2/masterfiles/$(branch)/inputs )
```

became this:

```
        AllowRedefinitionOf  = ( branch )
        branch          = ( PROD )

    dev_servers::
        branch          = ( "DEV/branches/1" )

    any::
        master_cfinput = ( /var/lib/cfengine2/masterfiles/$(branch)/inputs )
        master
```

And in `PROD/inputs/classes/cf.main_classes`, we added these lines:

```
    dev_servers              = (      ops1
                                      )
```

After all those updates are completed, we checked in the changes:

```
$ svn commit -m"support the DEV branch for the system ops1"
Sending        inputs/classes/cf.main_classes
Sending        inputs/control/cf.control_cfagent_conf
Sending        inputs/update.conf
Transmitting file data ...
Committed revision 11.
```

We're all set. Update the cfengine master with `svn update` in the `PROD` tree, and now *ops1* should be using the `DEV/branches/1` tree. There is one problem: the `DEV` tree hasn't been updated to point *ops1* at itself! This is the perfect opportunity to perform our first merge in Subversion.

MERGING CRASH COURSE

Most system administrators are familiar with the `diff` and `patch` commands. The `diff` command is used to compare text files line by line and show the differences. The `patch` command takes a file containing a difference listing produced by the `diff` program and applies those differences to one or more original files, producing patched versions. These are the traditional tools used to distribute and apply changes to files such as publicly available source code.

Revision control systems such as Subversion make the process of applying differences between files in the repository easier using facilities to merge the files. The merge process is essentially a `diff` and `patch` procedure done within the repository, complete with revision history of the merge operation. The advantage of merging over manual use of the `diff` and `patch` commands is that the history of the merged files will show exactly where the new file contents came from, including the specific revision and repository path of the source files.

   If merging is entirely new to you and you're still struggling to understand the concepts, you're not alone. Revision control system tools and concepts are best learned by working with them.

First, in our working copy, we ran `svn log` from the `DEV/branches/1` directory to note the revision at which we created the branch (revision 9):

```
$ cd ~/masterfiles/DEV/branches/1
$ svn log
------------------------------------------------------------------------
r9 | nate | 2008-09-01 03:27:26 -0700 (Mon, 01 Sep 2008) | 1 line

creating the first cfengine development branch
------------------------------------------------------------------------
```

The `svn log` output went on, but the first entry was the important one, because it was the last time that the branch was updated. The history beyond that point is actually the history of the PROD branch, because that's where the DEV branch was copied from. Up until that point there was *only* the PROD branch. We'll want everything done to the production branch from that point forward to be applied to the DEV branch—synchronizing the two branches completely.

   We then changed directory to the PROD directory to gather the latest revision of the PROD branch, since we'll want to apply everything done to the PROD branch since revision 9 back to the DEV branch. Then, we ran a merge as a dry run to see the files that have changed and would be merged. The commands to do this follow:

```
$ cd PROD/
$ svn status -u
Status against revision:     11
$ cd ../DEV/branches/1
$ svn  merge --dry-run -r 9:11 https://svn.campin.net/svn/cfengine/masterfiles/PROD
U    inputs/control/cf.control_cfagent_conf
U    inputs/update.conf
U    inputs/classes/cf.main_classes
U    repl/root/etc/bind/debian-ext/db.campin.net
```

This looks good, since those are the files with changes that need to be migrated over to the `DEV/branches/1` branch. We now need to go ahead and perform the merge against our working copy and inspect the changes:

```
$ svn merge -r 9:11 https://svn.campin.net/svn/cfengine/masterfiles/PROD \
-m"merging revisions 9-11 from PROD into DEV"
```

We inspected the changed files, and the expected changes are there. We'll commit our development branch with `svn commit` and update the `DEV/branches/1` tree on the cfengine master.

When merging, be sure to specify the revisions you're merging in the commit message, so that later, when you merge again, you can find the revision at which to start your new merge. You don't ever want to attempt to merge the same changes twice. The lack of detection and prevention of duplicated merges is an acknowledged weak spot in Subversion, and you don't want to get caught by it if you can avoid it.

MERGING FROM PROD TO DEV

Normally, when we merge between branches in our cfengine repository, we'll want the changes to be coming from the `DEV` tree and merged into the `PROD` tree. We want to test out changes in a nonproduction environment first. Sometimes, however, we won't have a suitable test environment and changes will go to the `PROD` tree first and will subsequently need to be merged back to `DEV` again.

Merging from `PROD` to `DEV` is okay here and there, but if you find yourself doing so on a regular basis, you probably need to invest in more hardware for your development environment. Either that, or you need to stop being so lazy and force yourself to test your changes first.

We'll be the first to admit, however, that there are some notable exceptions to the rule of testing first. Simple changes like DNS additions don't need to go through the testing environment when a sanity check like `named-checkzone` is utilized. The additional overhead and delay of pushing the change through the testing environment really isn't justified. Also, when a site is in its infancy stages, as ours is, there usually isn't the hardware and time yet to set up the `DEV` systems. Do yourself a favor, though, and get a number of systems running against the `DEV` tree as soon as possible. Testing changes there first might just save your job at some point.

Our host *ops1* is now utilizing a completely separate tree on the cfengine master, using a Subversion tree that we can leverage to share code between development and production. Setting up more hosts to use the `DEV` tree is as simple as adding hosts to the `dev_servers` class in `update.conf` and `inputs/classes/cf.main_classes` in both the `PROD` and `DEV` lines of development.

To make full use of the `DEV` tree, you'll want to specify a testing host for all of the production roles that you're using in the `PROD` tree, some of which follow:

- Debian Subversion, Nagios, and Ganglia web host

- Solaris NFS home directory server

- Red Hat public web server

- Debian DNS server

- Debian mail relay

Since we don't ever specify hostnames in the cfengine tasks, it's simply a matter of redefining some group memberships in the `DEV/branches/1/inputs/classes/cf.main_classes` file for testing purposes. Notice how abstracting the hostnames away from role names helps in yet another way. We're now free to test out entirely new DNS mechanisms or change anything else in our development environment, with no effect on production. Additionally, setting up virtual hosts under a system such as VMware can help ensure that not a lot of extra hardware is needed for testing purposes.

Note that we didn't cover usage of the `STAGE` directory tree. Our network is still small that we're not making use of that tree yet. The idea is that once our network is large enough, we'll have separate hosts for testing configurations once they come out of the initial development phase. Some changes might need days or weeks before they are approved for promotion to the main production branch. You can always use the `DEV` tree this way as well, but it's useful to give it a descriptive name such as `STAGE` if you intend to use it as a longer-term testing ground.

The usage of the `STAGE` tree will technically be identical to usage of the `DEV` tree. It is the policies around usage that will differ, and those need to be defined on a per-site basis.

Backups

A substantial amount of work has now been put into our cfengine master, as well as our three imaging systems. Since we set up Kickstart, Jumpstart, and FAI before we had cfengine managing our systems, we have no backups of those systems. In addition, we need to back up our cfengine Subversion repository. If we had automated the setup of the configuration of all three imaging system hosts with cfengine, we would need to back up only the Subversion repository.

We would like to have to back up only the Subversion repository. This would mean that all of the configuration at our site is performed via cfengine, which is how we want things. To use cfengine to perform all configuration at our site, we should go back and

automate the setup of our imaging systems as much as possible and then only back up Subversion.

The automation of our imaging systems would include neither the Kickstart and Jumpstart process of copying the installation image(s) to disk (`setup_install_server` for Jumpstart and the DVD copy to `/kickstart/rhel_5_2` on the Kickstart host) nor the installation client setup for those systems. We're looking to automate the synchronization of files that we had to manually create or manually edit.

Backing up only the Subversion repository obviously won't work for application data backups, but at this point, we don't have any application data to be concerned about. When we need to worry about application logs or other variable data, we'll want to investigate an open source backup solution such as Amanda or a commercial backup product such as Veritas NetBackup.

First, let's grab the important configuration files from our imaging systems, check them into Subversion, and distribute the files using cfengine.

Jumpstart

Jumpstart is great in that the setup is done entirely via scripts contained on the installation media. We don't need to worry about backing up most of the files in the `/jumpstart` directory tree. All we'll need to copy using cfengine is the `/jumpstart/profiles/` directory. Everything else that we need to re-create a functional Jumpstart server is contained in Chapter 6. Those steps don't lend themselves well to automation, since the steps to recreate the Jumpstart environment depend on having some form of installation media available—and it could be a series of CDs, a DVD, or an ISO file.

We copied the `/jumpstart/profiles` directory from our Jumpstart server *hemingway* into our working copy:

```
$ cd ~/masterfiles/PROD/repl/root/
$ mkdir jumpstart
$ scp -r root@hemingway:/jumpstart/profiles jumpstart/
```

Then, we added the `jumpstart` directory to the cfengine Subversion repository:

```
$ svn add jumpstart/
A         jumpstart
A         jumpstart/profiles
A         jumpstart/profiles/aurora
A         jumpstart/profiles/aurora/sysidcfg
A         jumpstart/profiles/aurora/finish_install.sh
A         jumpstart/profiles/aurora/rules
A         jumpstart/profiles/aurora/rules.ok
```

```
A          jumpstart/profiles/aurora/basic_prof
A          jumpstart/profiles/jumpstart_sample
A          jumpstart/profiles/jumpstart_sample/any_machine
A          jumpstart/profiles/jumpstart_sample/check
A          jumpstart/profiles/jumpstart_sample/host_class
A          jumpstart/profiles/jumpstart_sample/net924_sun4c
A          jumpstart/profiles/jumpstart_sample/rules
A          jumpstart/profiles/jumpstart_sample/set_nfs4_domain
A          jumpstart/profiles/jumpstart_sample/set_root_pw
A          jumpstart/profiles/jumpstart_sample/upgrade
A          jumpstart/profiles/jumpstart_sample/x86-begin
A          jumpstart/profiles/jumpstart_sample/x86-begin.conf
A          jumpstart/profiles/jumpstart_sample/x86-begin.conf/OWconfig
A          jumpstart/profiles/jumpstart_sample/x86-begin.conf/msm.conf
A          jumpstart/profiles/jumpstart_sample/x86-class
```

After that, we needed to distribute the profiles directory to the Jumpstart host. We created a class in cfengine for the role jumpstart_server, and added *hemingway* to that class. We used the class in a task located at PROD/inputs/tasks/app/jumpstart/cf.copy_ jump_profiles with these contents:

```
copy:
        jumpstart_server::
                $(master)/repl/root/jumpstart/profiles
                                dest=/jumpstart/profiles
                                mode=755
                                r=inf
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true


directories:
        jumpstart_server::
                /jumpstart/profiles mode=755 owner=root group=root inform=false
```

We copy all the files with mode 775, since some of them need to be executable. It won't hurt anything if they're all executable, just be aware that the executable bit being set in this directory doesn't mean that the file is necessarily a script.

We then added the PROD/inputs/tasks/app/jumpstart directory to the Subversion repository with this command:

```
$ pwd
/home/nate/masterfiles/PROD/inputs/tasks/app
$ svn add jumpstart/
A          jumpstart
A          jumpstart/cf.copy_jump_profile
```

Next, we added this line to PROD/inputs/classes/cf.main_classes to create the new class:

```
    jumpstart_server        = ( hemingway )
```

We then created a hostgroup file for the jumpstart_server class, with a new file at the location PROD/inputs/hostgroups/cf.jumpstart_server with these contents:

```
import:
     any::
              tasks/app/jumpstart/cf.copy_jump_profiles
```

Be sure to svn add the cf.jumpstart_server file into the repository.

As usual, the last step is to set up the cfengine import of this hostgroup file in the hostgroup mapping file at PROD/inputs/hostgroups/cf.hostgroup_mappings. We added this line:

```
    jumpstart_server::              hostgroups/cf.jumpstart_server
```

Since this was all done in our working copy, we needed to check in all these changes.

```
$ cd ~/masterfiles/PROD/
$ svn commit \
-m"set up the copy of the jumpstart profiles directory to the jumpstart host"
```

We then checked them out on the cfengine master with the svn update command in the PROD directory.

We should now be able to restore what we need, if and when the *hemingway* host dies and is subsequently reinstalled. All the rest of the configuration on the host is easily reproducible, simply by referring to the Jumpstart section in Chapter 6.

Kickstart

To distribute the important files that would need restoration if the Kickstart host is rebuilt, we first copied the important files into our cfengine repository working copy:

```
$ cd ~/masterfiles/PROD/repl/root/
$ mkdir -p kickstart/rhel5_2
$ scp -r root@rhmaster:/kickstart/cfengine-2.2.7 kickstart/
$ scp -r root@rhmaster:/kickstart/scripts kickstart/
$ scp  root@rhmaster:/kickstart/rhel5_2/ks.cfg kickstart/rhel5_2/
$ svn add kickstart
```

After that, we needed to copy out these files to the /kickstart directory on the host *rhmaster* using cfengine. Once again in our working copy, we created the directory PROD/inputs/tasks/app/kickstart, and created a task in the directory called cf.copy_kickstart_dir with these contents:

```
copy:
        kickstart_server::
                $(master)/repl/root/kickstart
                                dest=/kickstart
                                mode=755
                                r=inf
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true


directories:
        kickstart_server::
                /kickstart mode=755 owner=root group=root inform=false
```

We added the PROD/inputs/tasks/app/kickstart directory to Subversion with svn add once we had the task file inside it. Next, we needed to do the usual steps in order to make this task get used by our Kickstart server. Here's a summary of the steps:

1. Create the kickstart_server class in PROD/inputs/classes/cf.main_classes.

2. Create the hostgroup file at PROD/inputs/hostgroups/cf.kickstart_server that imports the cf.copy_kickstart_dir task. Add the file to the Subversion repository.

3. Set up the hostgroup import in the hostgroup mapping file PROD/inputs/hostgroups/cf.hostgroup_mappings.

4. Commit the changes to your working copy, and update the production working copy on the cfengine master.

Now our important Kickstart files are contained in Subversion and will be restored by cfengine via a copy if we ever have to rebuild our Kickstart server.

FAI

When we set up FAI, we were careful to modify the default FAI configuration files as little as possible. We wanted to be able to push new files as much as possible, since we knew that we would want to distribute those files using cfengine later on.

We collected all the files under the /srv/fai/config directory that we modified or added back in Chapter 6 in our working copy of the repository:

```
$ pwd
/home/nate/masterfiles/PROD/repl/root/srv/fai/config
$ ls -R
.:
./  ../  class/  disk_config/  files/  hooks/  package_config/  scripts/

./class:
./  ../  60-more-host-classes*  FAIBASE.var

./disk_config:
./  ../  LOGHOST  WEB

./files:
./  ../  etc/

./files/etc:
./  ../  cfengine/

./files/etc/cfengine:
./  ../  cfagent.conf/  update.conf/

./files/etc/cfengine/cfagent.conf:
./  ../  FAIBASE*

./files/etc/cfengine/update.conf:
./  ../  FAIBASE*

./hooks:
./  ../  savelog.LAST.source*
```

```
./package_config:
./  ../  FAIBASE  LOGHOST  WEB

./scripts:
./  ../  FAIBASE/

./scripts/FAIBASE:
./  ../  50-cfengine*  60-create-cf-config*
```

We'll distribute all these as another recursive copy, this time into the `/srv/fai/config` directory on the FAI server (*goldmaster*). We have some additional files that we modified during the setup of our FAI server:

- `/etc/fai/make-fai-nfsroot.conf`

- `/etc/dhcp3/dhcpd.conf`

- `/etc/inetd.conf`

There is a problem with `/etc/inetd.conf`: in the task `PROD/inputs/tasks/app/rsync/cf.enable_rsync_daemon`, we add a line to `/etc/inetd.conf` using the `editfiles` action. This `editfiles` action must be changed or removed, since it makes no sense to have an `editfiles` action acting on a file that cfengine is also copying out. Two scenarios could result, depending on the contents of the `inetd.conf` file that cfengine copies into place:

- The copied `/etc/inetd.conf` file won't have the entry that the task `cf.enable_rsync_daemon` is looking for, and it will be added by the `editfiles` action. This means that the next time cfengine runs, `/etc/inetd.conf` won't match the checksum of the file in the `masterfiles` tree, and `inetd.conf` will be copied again. After that, the `editfiles` action will once again notice that the required entry isn't there, and it will add it yet again. This loop will continue on every time cfengine runs.

- The copied `/etc/inetd.conf` file will already have the required entry, making the `editfiles` action unnecessary.

You can see that, either way, we don't need the `editfiles` action. It either produces what we can only consider an error by constantly changing the file or is totally unneeded. We'll simply place the required entry in the `inetd.conf` file that we copy out and remove the `editfiles` section from the `cf.enable_rsync_daemon` task. We *will* add a comment to the task, however, stating that the enable of the daemon is handled via a static file copy in another task and provide the task file name in the comment.

After editing the `PROD/inputs/tasks/app/rsync/cf.enable_rsync_daemon` task to comment out the `editfiles` section and add the new comment, we placed these files into our working copy of the cfengine tree:

```
$ pwd
/home/nate/masterfiles/PROD/repl
$ cp /etc/inetd.conf root/etc/
$ svn add root/etc/inetd.conf
A         root/etc/inetd.conf
$ cp /etc/fai/make-fai-nfsroot.conf root/etc/fai/
$ svn add root/etc/fai/make-fai-nfsroot.conf
A         root/etc/fai/make-fai-nfsroot.conf
$ mkdir root/etc/dhcp3
$ cp /etc/dhcp3/dhcpd.conf root/etc/dhcp3/
$ svn add root/etc/dhcp3
A         root/etc/dhcp3
A         root/etc/dhcp3/dhcpd.conf
```

Note that the copies were local since we were working in our home directory from the *goldmaster* system itself.

We created a task at PROD/inputs/tasks/app/fai/cf.copy_fai_files with these contents:

```
control:
        fai_server::
                AddInstallable          = ( restart_inetd restart_dhcpd )


copy:
        fai_server::
                $(master)/repl/root/srv
                                dest=/srv
                                mode=755
                                r=inf
                                owner=root
                                group=root
                                type=checksum
                                server=$(fileserver)
                                encrypt=true

                $(master_etc)/inetd.conf
                                dest=/etc/inetd.conf
                                mode=755
                                owner=root
                                group=root
```

```
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
                              define=restart_inetd

                $(master_etc)/fai/make-fai-nfsroot.conf
                              dest=/etc/fai/make-fai-nfsroot.conf
                              mode=755
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true

                $(master_etc)/dhcp3/dhcpd.conf
                              dest=/etc/dhcp3/dhcpd.conf
                              mode=755
                              owner=root
                              group=root
                              type=checksum
                              server=$(fileserver)
                              encrypt=true
                              define=restart_dhcpd

directories:
        fai_server::
                /srv mode=755 owner=root group=root inform=false

shellcommands:
        debian.restart_inetd::
                "/etc/init.d/openbsd-inetd restart" timeout=30 inform=true

        debian.restart_dhcpd::
                "/etc/init.d/dhcp3-server restart" timeout=30 inform=true
```

We made sure to add the new `tasks/app/fai` directory to the repository. We need to create the `fai_server` class, create a `hostgroup` file for it, and import it in the `cf.hostgroup_mappings` file. Here's a summary of the steps:

1. Create the `fai_server` class in `PROD/inputs/classes/cf.main_classes`.

2. Create the hostgroup file at `PROD/inputs/hostgroups/cf.fai_server` that imports the `cf.copy_fai_files` task. Add the file to the Subversion repository.

3. Set up the hostgroup import in the hostgroup mapping file `PROD/inputs/hostgroups/cf.hostgroup_mappings`.

4. Commit the changes to your working copy, and update the production working copy on the cfengine master.

Subversion Backups

The procedure to back up a Subversion repository is quite simple. We can use the `svnadmin` command with the `hotcopy` argument to properly lock the repository and perform a file-based backup. Backing up this way is much better than performing a `cp` or `rsync` copy of the repository files, which might result in a corrupted backup.

Use the command like this:

```
# svnadmin hotcopy /path/to/repository /path/to/backup-repository
```

The repository made by `svnadmin hotcopy` is fully functional; we are able to drop it in place of our current repository should something go wrong. We can create periodic backups of our repository this way and copy the backups to another host on our network or even to an external site.

Be aware that each time a hot copy is made, it will use up the same amount of disk space as the original repository. Backup scripts that make multiple copies using `svnadmin hotcopy` will need to be careful not to fill up the local disk with backups.

We'll create a script at `PROD/repl/admin-scripts/svn-backup` with these contents (explained section by section):

```
#!/bin/sh
# This script is tested on Debian Linux only.
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/opt/admin-scripts

SVN_REPOS="/var/svn/repository/binary-server /var/svn/repository/cfengine"

case `hostname` in
etchlamp*)
        echo "This is the host on which to backup the Subversion repo, continuing."
        ;;
*)
```

```
        echo "This is NOT the host on which to backup the SVN repo, exiting..."
        exit 1
        ;;
esac
```

Since we copied the script to all hosts on our network, we took steps to make sure that it only runs on the proper host:

```
BACKUP_BASE_DIR=/var/backups
LOCKFILE=/root/svn_backup_lock

rm_lock_file() {
        rm -f $LOCKFILE
}
```

We'll be using file locking to prevent two invocations of this script from running at once.

```
rotate_backups() {
        BACKUP_DIR_NAME=$1

        if cd $BACKUP_DIR_NAME
        then
                for num in 6 5 4 3 2 1
                do
                        one_more=`expr $num + 1`
                        if [ -d backup.${num} ]
                        then
                                if [ -d backup.${one_more} ]
                                then
                                        rm -rf backup.${one_more} && \
                                        mv backup.${num} backup.${one_more}
                                else
                                        mv backup.${num} backup.${one_more}
                                fi
                        fi
                done
        else
                echo "Can't cd to $BACKUP_DIR_NAME - exiting now"
                rm_lock_file
                exit 1
        fi
}
```

We wrote a subroutine to manage our stored backup directories. It takes an argument of a repository directory that needs to be backed up, and it moves any numbered backup directories to a new backup directory with the number incremented by one. A backup directory with the number 7 is removed, since we only save seven of them.

For example, the directory `/var/backups/binary-server/backup.7/` is removed, and the directory `/var/backups/binary-server/backup.6/` is moved to the name `/var/backups/binary-server/backup.7/`. The subroutine then progresses backward numerically from 5 to 1, moving each directory to another directory with the same name except the number incremented by 1. When it is done, there is no directory named `/var/backups/binary-server/backup.1/`, which is the directory name we'll use for a new Subversion backup:

```
# don't ever run two of these at once
lockfile $LOCKFILE || exit 1

for REPO in $SVN_REPOS
do
        SHORTNAME=`basename $REPO`
        BACKUP_DIR="$BACKUP_BASE_DIR/$SHORTNAME"
        [ -d "$BACKUP_DIR" ] || mkdir -p $BACKUP_DIR

        cd $BACKUP_DIR && rotate_backups $BACKUP_DIR

        /usr/bin/svnadmin hotcopy $REPO $BACKUP_DIR/backup.1
done
```

In this section, we perform these steps:

1. Retrieve just the short portion of the directory name using the `basename` command so that the variable `SHORTNAME` contains the value `binary-server` or `cfengine`—the two repository directory names.

2. We then make sure that the directory used for the backups exists and create it if necessary.

3. Now that the directory is known to exist, we change directory to the proper backup directory and use our subroutine that rotates the previous backup directories.

4. Then we use the `svnadmin hotcopy` command to create a new backup of the repository. This is done for each directory listed in the variable `SVN_REPOS`.

```
# if we get here without errors, clean up
rm_lock_file
```

Finally, we removed the lock file that is used to prevent two of these from running at once. We ran the script eight times in a row to demonstrate the output, here it is:

```
# hostname
etchlamp
# ls -ltr /var/backups/binary-server/
total 28
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.7
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.6
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.5
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.4
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.3
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.2
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.1
# ls -ltr /var/backups/cfengine/
total 28
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.7
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.6
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.5
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.4
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.3
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.2
drwxr-xr-x 7 root root 4096 2008-09-01 23:31 backup.1
```

In order to use the `lockfile` command (contained in the script), the package `procmail` needs to be installed. Add the string `procmail` on a line by itself to your working copy of `PROD/repl/root/srv/fai/config/package_config/FAIBASE`, and check in the modification so that all future hosts get the package installed. For now, just install the `procmail` package using `apt-get` on the Subversion sever (the system *etchlamp*).

We'll create a task to run the backup script once per day, in a file at the location `PROD/inputs/tasks/app/svn/cf.svn_backups` with these contents (be sure to add it into the Subversion repository):

```
shellcommands:
        svn_server.debian.Hr00.Min00_05::
                "/opt/admin-scripts/svn-backup"
                        timeout=600
```

We're using cfengine to run the backups every day between midnight and five minutes after midnight. Remember that we set a five-minute `SplayTime`, so `cfagent` will run

at some time in the five minutes after midnight. We need to specify the range so that our `shellcommands` action will run. The absolute time class of `Min00` probably wouldn't match, but the range `Min00_05` definitely will.

Now, we need to add this line to `PROD/inputs/hostgroups/cf.svn_server`:

```
tasks/app/svn/cf.svn_backups
```

Commit your changes to the repository, and update the production working copy. Now, every night at midnight, a new backup will be created, and we'll always have seven day's worth of backups on hand.

Copying the Subversion Backups to Another Host

We will copy the Subversion backup directories to another host on our local network using cfengine, so we'll be able to quickly restore our two Subversion repositories if the Subversion server fails.

We'll modify our site's shared `cfservd.conf` configuration file to grant access to the backup directories on *etchlamp* from a designated backup host. We will use the cfengine master as the backup host and always keep a complete backup of those directories.

We added these lines to `PROD/inputs/cfservd.conf` in the `admit:` section:

```
etchlamp::
        # Grant access to the Subversion backups to the goldmaster host
        /var/backups/binary-server    192.168.1.249
        /var/backups/cfengine             192.168.1.249
```

Then, we created a task to copy the directories, the file `PROD/inputs/tasks/app/svn/cf.copy_svn_backups` with these contents (and we added the file to the repository, of course):

```
copy:
     fileserver.Hr00.Min20_25::
          /var/backups/cfengine
                         dest=/var/backups/svnbackups/cfengine
                         mode=555
                         r=inf
                         purge=false
                         owner=root
                         group=root
                         type=checksum
                         server=$(svn_server)
                         encrypt=true
                         trustkey=true
```

```
                /var/backups/binary-server
                                dest=/var/backups/svnbackups/binary-server
                                mode=555
                                r=inf
                                purge=false
                                owner=root
                                group=root
                                type=checksum
                                server=$(svn_server)
                                encrypt=true
                                trustkey=true


directories:
        policyhost::
                /var/backups/svnbackups/cfengine mode=750
                        owner=daemon group=root inform=false

                /var/backups/svnbackups/binary-server mode=750
                                owner=daemon group=root inform=false
```

We then added this line to `PROD/inputs/control/cf.control_cfagent_conf` so that we could abstract the hostname of the Subversion server with a variable:

```
svn_server      = ( etchlamp.campin.net )
```

Next, we added a comment to `PROD/inputs/classes/cf.main_classes` so that this line:

```
    svn_server                 = ( etchlamp )
```

became this:

```
    # we also define svn_server as a variable in the file
    # inputs/control/cf.control_cfagent_conf - update that file
    # as well if you change the svn_server class below.
    svn_server                 = ( etchlamp )
```

We then needed a hostgroup file for the `policyhost` machine, so we created `PROD/inputs/hostgroups/cf.policyhost` with these contents:

```
import:
        any::
                tasks/app/svn/cf.copy_svn_backups
```

And we added this line to `PROD/inputs/hostgroups/cf.hostgroup_mappings`:

```
policyhost::                    hostgroups/cf.policyhost
```

Commit your changes, and update the production `PROD` tree on the cfengine master. The next day (after 12:25 a.m.), you should have fully functional Subversion backups stored in the `/var/backups/svnbackups/` directory on your cfengine master.

We'll leave the task of copying the backup directories to an offsite host as an exercise for you.

Enhancement Is an Understatement

This chapter took our site from being at a high risk due to system failure to being a fully version controlled and backed up environment.

Many sites that utilize cfengine or other automated management software don't have the ability to easily manage a testing environment such as the one demonstrated here. We have a real advantage in the existence of our `DEV` cfengine branch, and we should use it as much as possible to try out new configurations and applications.

Our backup measures are certainly minimal, but they're effective. If we suffered total system failure on any of our hosts, including the critical cfengine master, we can restore the system to full functionality.

■ ■ ■

Improving System Security

Early in this book, we established that managing the contents and permission of files is the core of UNIX/Linux system administration. UNIX/Linux security is also almost entirely concerned with file contents and permissions. Even when configuring network settings for security reasons, we're usually configuring file contents. This means that, in general, we'll be performing very familiar operations when using cfengine to increase the security of our UNIX and Linux hosts.

At various points in this book, we've taken security into account when configuring our systems or when implementing some new functionality

- We centralized our user account files right away in our example site, in order to easily change passwords and add and remove accounts across our site.

- We run all of our internal web sites over HTTPS only (Nagios, Ganglia, and Subversion).

- We compiled our own Apache from source so that our externally facing web site has the fewest features possible, which should decrease the likelihood of our site being vulnerable to remote Apache exploits.

- We don't allow root privileges over NFS.

- We set up a central log host along with automated log reporting.

- We made sure our centralized `cfexecd` log uploads were protected against malicious users.

- We configured version control and backups at our site. This may seem like more of a disaster recovery measure, but modern data security is just as concerned with a disaster destroying information as it is about damage from attackers.

In this chapter, we focus on security itself, but we don't mean to give you the idea that security is a separate duty from your normal ones. If treated as an afterthought, good security is difficult to obtain and, in fact, becomes something of a burden if addressed during the later phases of a project.

Since we're working only on the hosts on our network, we're addressing host-based security. We feel that the importance of host-based security measures cannot be overstated. Many sites implement network security through the use of firewalls and put very little work into the security of the hosts on their network. Such an approach assumes (or naively hopes) that no threats exist on the internal network. Most firewalls by their very nature allow particular traffic through to hosts on the internal network. This traffic could be utilized by attackers to compromise internal hosts, which can then be used as a jumping off point to attack other hosts.

We need to remember that internal users are a major risk. Even if the users themselves aren't malicious, their credentials or their computer systems can be compromised and used by attackers to access the internal network via a VPN or other remote access methods. No modern network should have a crunchy exterior and a chewy interior—meaning perimeter network protection without internal protection mechanisms.

Host-based security mechanisms go a long way toward hardening the internal network. Shutting down unneeded daemons, removing unnecessary accounts, removing or minimizing trust between hosts, implementing proper file permissions and host-based firewalls, and frequently applying system patches and updated packages will address the vast majority of local and remote vulnerabilities.

■**Note** As you might guess, we can't provide a comprehensive security guide in just one chapter. What we can do, however, is recommend the book *Practical UNIX & Internet Security* by Simson Garfinkel, Alan Schwartz, and Gene Spafford (O'Reilly Media Inc., 2003).

Security Enhancement with cfengine

Cfengine can improve system security in many ways. First, it allows you to automatically configure systems in a consistent manner. The cfengine configuration is general enough that you can quickly apply your changes to other hosts in the same or different classes, even to systems that haven't been installed yet. This means that if you correct a security problem on your Linux systems through cfengine, and then later install a new Linux system, the security problem will be fixed there as well (if necessary).

Some other ways cfengine can help with system security are illustrated within the following sections. Just be aware that this is far from a comprehensive list. Your own systems will almost certainly have more areas where you can use cfengine to enhance their security. You may choose to run applications like FTP servers that can be serious security problems if not properly configured. We can't cover all of these situations, but a good security book will tell you what to configure, and cfengine can do the actual configuration for you.

As always, we do all of our system administration in our example infrastructure using cfengine, so this final chapter doesn't look all that different from the earlier ones. The difference here is that we're not focusing much on the cfengine configuration but more on the security gains from the changes we make.

Removing the SUID Bit

One of the most common ways for a malicious user to gain privileged access is via flaws in programs with the setuid (or SUID) bit set. This permission setting causes a program to be executed with the privileges of the file's owner, not those of the user executing the program. It is a UNIX mechanism that allows nonprivileged users to perform tasks that require elevated privileges (usually, though not always, root privileges). A programming error or flaw in such a program is often disastrous to local security. The two ways to avoid becoming a victim of such a flaw are to keep your system up to date with security and bug fixes and to limit the number of setuid binaries on your system that are owned by the root user.

We should first give you an idea of what SUID binaries are present on our systems, which will allow us to make educated decisions about what to exclude from a file sweep that removes the SUID bit. The following `find` command will work on all systems at our example site, should be run as `root`, and allows us to view the list and determine what to allow:

```
# find / -fstype nfs -prune -o -user root -perm -04000 -ls | tee /var/tmp/suid.list
```

This `find` command will not descend into filesystems mounted over NFS and will find programs owned by the root user that have the SUID bit set. It then uses the `tee` command to save the output into a file for later investigation, while still displaying the output to the screen.

On our Debian systems, which we imaged with FAI and configured via cfengine, the output was rather short, a total of 20 programs. Part of the reason for that is because we haven't installed the X Window System, but it mainly reflects a very security-conscious Linux distribution.

On our Solaris system imaged with Jumpstart, we got an astonishingly long list, with 75 total entries.

On the Red Hat system that we imaged via Kickstart, we found 36 SUID root-owned files, which also isn't too bad. Kudos go to Red Hat for cleaning up the situation; in the past, Red Hat was one of the worst offenders among Linux distributions.

To remove the SUID bit from all the binaries except those that we deemed important, we created a task at `PROD/inputs/tasks/os/cf.suid_removal` with these contents:

```
files:
      debian.Hr03.Min40_45::
            /
                  filter=rootownedfiles
                  mode=-4000        # no SUID for rootownedfiles
                  recurse=inf
                  action=fixall
                  inform=true
                  ignore=/usr/bin/passwd
                  ignore=/usr/bin/traceroute.lbl
                  ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_icmp
                  ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_dhcp
                  ignore=/usr/lib/pt_chown
                  ignore=/sbin/unix_chkpwd
                  ignore=/bin/ping
                  ignore=/bin/su
                  syslog=on
                  xdev=on


      redhat.Hr03.Min40_45::
            /
                  filter=rootownedfiles
                  mode=-4000        # no SUID for rootownedfiles
                  recurse=inf
                  action=fixall
                  inform=true
                  ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_dhcp
                  ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_icmp
                  ignore=/usr/bin/sudo
                  ignore=/usr/bin/crontab
                  ignore=/usr/bin/at
                  ignore=/usr/bin/sudoedit
                  ignore=/usr/sbin/ccreds_validate
                  ignore=/bin/ping
                  ignore=/bin/su
                  ignore=/sbin/unix_chkpwd
                  ignore=/sbin/pam_timestamp_check
                  syslog=on
                  xdev=on
```

```
(solaris|solarisx86).Hr03.Min40_45::
        /
                filter=rootownedfiles
                mode=u-s         # no SUID
                recurse=inf
                action=fixall
                inform=true
                ignore=/proc
                ignore=/opt/csw/bin/sudo.minimal
                ignore=/opt/csw/bin/sudo
                ignore=/opt/csw/bin/sudoedit
                ignore=/usr/bin/at
                ignore=/usr/bin/atq
                ignore=/usr/bin/atrm
                ignore=/usr/bin/crontab
                ignore=/usr/bin/su
                ignore=/usr/lib/pt_chmod
                ignore=/usr/lib/utmp_update
                ignore=/usr/sbin/traceroute
                ignore=/usr/sbin/ping
                ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_dhcp
                ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/check_icmp
                ignore=/usr/pkg/nagios-plugins-1.4.12/libexec/pst3
                syslog=on
                xdev=on
```

We set xdev=on so that cfengine doesn't cross filesystem boundaries. We know that we imaged all of our systems with a single root filesystem, so this keeps us from crawling the NFS directories. Even if we wanted to fix the permissions on NFS mounts, we couldn't because the root user is mapped to the nobody user over NFS (unless the no_root_squash option is used on the NFS server, which we don't use; refer to the NFS section in Chapter 8).

We utilized the rootownedfiles filter from the file PROD/inputs/filters/cf.root_owned, which is imported from cfagent.conf. The file has these contents:

```
filters:

        { rootownedfiles

        Owner: "root"

        Result: "Owner"
}
```

Filters in cfengine can get very complicated and are able to look for several items with particular attributes in order to successfully match. The preceding filter is a very simple file one that matches when a file is owned by root. In conjunction with these lines from `cf.suid_removal`

```
mode=u-s          # no SUID
recurse=inf
action=fixall
```

we tell cfengine that we want the files to lack the SUID bit, that cfengine should infinitely recurse directories, and that the action to take is to fix the files. The final setting is to ignore the files that we don't want changed, using the `ignore=` lines.

To activate this task, we added this line to `PROD/inputs/hostgroups/cf.any`:

```
tasks/os/cf.suid_removal
```

Be careful to test out these changes on just one host of each platform. As a temporary measure, you can override the hostgroups mechanism with lines like these in `PROD/inputs/hostgroups/cf.any`:

```
aurora|rhlamp|loghost1::
    tasks/os/cf.suid_removal


any::
```

Just be sure to set the `any::` class again at the end, since any entries added below later on will apply only to the three hosts specified. It will help avoid issues if another task needs to be imported to all hosts but is erroneously only imported for the three hosts mentioned previously. We don't want to leave an entry like this in place long term, since it circumvents our `hostgroups` cfengine configuration file organization method. Anytime that you specify hostnames as classes directly in any sort of actions, even imports, you're making it harder to maintain your infrastructure. Sticking with role-based classes aids maintainability in the long term. Ideally, hostnames should only show up in class definitions.

When new software is installed, be aware that if it installs a root-owned program with the setuid bit set, that the software may break due to this nightly-run task. We consider this a feature, not a bug. No new programs will last more than a day with the setuid bit set on our systems.

Protecting System Accounts

Standard system accounts are commonly used for brute force login attempts to systems. Every day, lists of common system accounts along with common passwords are used to attempt unauthorized logins by attackers.

We can protect ourselves against such attacks in three ways:

- Set system accounts to use nonworking shells.

- Remove unneeded system accounts.

- Lock the system accounts passwords.

We will attempt to make the system accounts on our systems unusable for interactive login. We have already set up our new accounts (such as the ganglia user) to not have a valid shell:

```
ganglia:x:106:109:Ganglia Monitor:/var/lib/ganglia:/bin/false
```

We need to duplicate this shell entry for all system accounts, with the notable exception of the root account.

■Note In the past, we've observed problems with daemons that utilized su - ACCOUNT in start-up scripts. If a daemon or script tries to execute a login shell this way, it won't function in our environment. Such start-up scripts don't require us to give the account a working shell, we can simply modify the script to use the -s /bin/sh option to su in order to make them work.

Since we have automated the distribution of centralized /etc/passwd files in our environment, we simply need to edit the copies in our Subversion DEV repository and test on some nonproduction hosts. We feel that an extra level of caution is needed with such changes. Once tested, merge the changed passwd files back to the PROD branch, and perform a Subversion check out in the production working copy on your cfengine master.

While editing the system accounts to change the shell to /bin/false, remove any accounts that aren't needed at your site. This may take some trial and error and should also be tested in a nonproduction environment before the changes are used in the PROD branch.

Next, edit the shadow files for all your site's platforms. Make sure that each account's encrypted password entry has an invalid string:

```
nagios:!:14115:0:99999:7:::
```

The bang (!) character in the encrypted password field of the `nagios` user account is an invalid string, locking the account. You can validate this with the `-S` argument to the `passwd` command on Linux:

```
$ sudo passwd -S nagios
nagios L 08/24/2008 0 99999 7 -1
```

The `L` in the output shows that the account is locked. This is the desired state for all our system accounts (besides the root account, of course). On Solaris the `-s` argument is used:

```
$ sudo passwd -s nagios
nagios     PS     08/24/08     0   99999      7
```

The `PS` field denotes either "passworded" or "locked," but we know our `nagios` account has no valid password! The Solaris `passwd` command expects a particular string in the encrypted password in order for it to report the LK (locked) status—the string `*LK*`. We can leave the account with just the bang and know that we're safe even through the Solaris `passwd` command doesn't understand it.

Applying Patches and Vendor Updates

Both Debian and Red Hat distributions make keeping systems up to date extremely easy with security patches and bug fixes. When using Red Hat Enterprise or the stable Debian branch (as we are), automatically updating system software is quite safe. Simple `shellcommands` sections that execute these commands will keep your Debian and Red Hat Enterprise systems fully patched and up to date:

- *Red Hat*: `# /usr/bin/yum upgrade`

- *Debian*: `# /usr/bin/apt-get update && /usr/bin/apt-get upgrade`

Solaris is another matter entirely. At the shops where we work full time, we still utilize Sun-recommended patch clusters and install them on a per-system basis in single-user mode. Every Sun tool that claims to automate system patches has either not worked to our satisfaction or required major infrastructure changes to accommodate the suite of Sun tools that are required. We find it useful to have a console connection to view the patch cluster output before attempting a system reboot, as serious problems have resulted that don't allow a proper reboot without prior repair.

One of the wisest ways to patch Sun systems is probably the Sun Live Upgrade procedure, where a patched Solaris operating system is installed to an alternate slice on a system's disks, and the host is then booted into the newly patched OS. If there are problems, the system can be booted back into the original OS install and full functionality is restored.

This approach requires some planning at initial installation time, since unused space needs to be left on the drives. The system's swap slice can be used, but this method isn't ideal, since the system is deprived of swap space and the swap slice often isn't large enough to hold a complete Solaris installation.

At the time of this writing, we recommend Live Upgrade and look forward to developing a proper automated mechanism for the third edition of this book.

Shutting Down Unneeded Daemons

Programs that accept network connections are like a door into your systems. Those doors might be locked, but most doors—like many network-enabled daemons—can be forced open. If you don't need the program, it should be shut down to reduce the overall exposure of your systems to network-based intrusion.

In this section, we will develop a task that shuts down a single service on each of the platforms in our example infrastructure to give you an example of how to do it on your own. Please carefully examine all running processes on your systems, and where possible, you should disable the unneeded daemons at installation time. We will write our cfengine task in such a way that if the programs aren't enabled, cfengine will do nothing.

We placed a task at `PROD/inputs/tasks/os/cf.kill_unwanted_services` with these contents:

```
control:
        any::
                AddInstallable         = ( disable_xfs )

processes:
        solarisx86|solaris::
                "dtlogin" signal=kill

        redhat::
                "xfs" action=warn matches=<1 define=disable_xfs

shellcommands:
        redhat.disable_xfs::
                "/sbin/service xfs stop"  timeout=60 inform=true
                "/sbin/chkconfig xfs off" timeout=60 inform=true

disable:
        solarisx86|solaris::
                /etc/rc2.d/S99dtlogin
```

We chose to shut down two different daemons used for the X Window System. On Solaris, the `dtlogin` daemon handles graphical logins, which we don't need on our server systems. On Red Hat, the `xfs` daemon is the X font server, also not needed on our server systems.

Fortunately for our security, but unfortunately for this book, none of our Debian systems was running any unneeded daemons. Going off the examples here and the experience gained so far in this book, you shouldn't have a trouble working out how to shut down Debian services. It could be done the same way the Solaris `dtlogin` daemon is shut down, via a process kill along with a disable of the start-up script.

We added the `cf.kill_unwanted_services` task to the `cf.any` hostgroup, checked in our changes, and updated the `PROD` tree on the cfengine master.

Removing Unsafe Files

You can use cfengine to disable a variety of files and programs on your system (if they exist). When executables and any other files are disabled, they are renamed with a `.cf-disabled` extension and their permissions are set to `0400`. In our example environment, we use a global backup directory (`$workdir/backups`), so the files are moved there for long-term storage.

Here is an example:

```
disable:
    any::
        /root/.rhosts       inform=true
        /etc/hosts.equiv inform=true

        # SunOS / NSDAP Rootkit
        /usr/lib/vold/nsdap/.kit           inform=true
        /usr/lib/vold/nsdap/defines     inform=true
        /usr/lib/vold/nsdap/patcher     inform=true
```

This disables the files `/root/.rhosts` and `/etc/hosts.equiv` on all systems (class `any`) because using these files is often considered a security risk. We also remove some files that result from the installation of an old rootkit. Rootkits are ready-to-run code made available on the Internet for attackers to maintain control of compromised hosts.

The `inform=true` entries will result in `cfagent` sending a message to standard output if and when it disables the files. This message will show up in `cfexecd` e-mails, as well as in the cfoutputs and syslog reports (see Chapter 9). Here's an example pair of syslog entries (one for the file rename and one for the move to the cfengine backup repository):

```
Sep 23 01:52:04 aurora cfengine:aurora[10573]: [ID 702911 daemon.notice]
Disabling/renaming file /etc/hosts.equiv to /etc/hosts.equiv.cfdisabled
(pending repository move)

Sep 23 01:52:04 aurora cfengine:aurora[10573]: [ID 702911 daemon.notice] Moved
/etc/hosts.equiv.cfdisabled to repository location
/var/cfengine/backups/_etc_hosts.equiv.cfdisabled
```

■**Note** Removing the example rootkit files with cfengine's `disable` action doesn't remove a rootkit from your system. Look into rootkit detection programs such as chkrootkit. If you confirm that a rootkit is installed on one of your systems, remove the system from the network, retrieve any important data, and reimage the host. The follow-on actions are to confirm that your data isn't compromised, that the attacker isn't on any of your other systems, and that your system is secured after reimaging (preferably during reimaging) so that the attacker doesn't get back in again.

File Checksum Monitoring

You can also use cfengine to monitor binary files on your system. Like any other file, the permissions of a binary file can be checked and any problems can be fixed. For binaries, particularly those of the setuid root variety, this feature can be very useful. You can also use cfengine to provide some tripwire functionality: you can use it to monitor the MD5 checksum of a file. Here is an example:

```
files:
    /bin/mount mode=4555 owner=root group=root action=fixall checksum=md5
```

On many systems, the `/bin/mount` program has the setuid bit set and is owned by the `root` user. This allows normal users to mount specific drives without superuser privileges. The parameters given in this example tell cfengine to check the permissions on this binary (and all others that are setuid root) and to record its checksum in a database.

If the checksum does change, you will be notified every time `cfagent` runs. This notification will continue until you execute `cfagent` with the following setting in the `control` section:

```
control:
    ChecksumUpdates = ( on )
```

This setting will cause all stored file checksums to be updated to their current values.

Using the Lightweight Directory Access Protocol

The Lightweight Directory Access Protocol (LDAP) allows you to use a central information repository for a variety of system and application uses. Although just about any information can be stored in an LDAP server, the most common thing to store is your user account information. For each user, you can specify an account, full name, phone number, office location, and any other information you may need.

Using LDAP for user directory and authentication at your site can increase your site's overall security, because a centralized authentication directory service enables the following:

- You can set up user account lockout when a user has a certain number of failed logins across one or many systems. If the lockout settings are local to each system, an attacker can attempt guesses against all systems at your site before the account is totally locked out.

- Passwords can be centralized across more applications than just UNIX/Linux logins, which allows the administrator to enable a single sign-on infrastructure. The administrators can then enforce strong password policies in this centralized directory.

We already have user account information at our example site centralized in the account files on our cfengine master. We have many of the benefits of using LDAP for centralized authentication, such as easy account auditing, easy password changes, and unified user IDs across all systems.

Any LDAP-aware application can retrieve data from the LDAP server. The Apache web server, for example, can use this information when it is authenticating users who are visiting a restricted web site. It is even more common to use LDAP to store the actual user accounts for your systems. Your operating system can probably use a remote LDAP server in addition to the local user list (`/etc/passwd`), since most modern UNIX systems support Pluggable Authentication Modules (PAM).

If your system does not come with an LDAP server or you need additional LDAP clients, take a look at OpenLDAP (`http://www.openldap.org/`). It provides an LDAP server as well as client libraries and compiles on a wide variety of systems. A second, newer alternative is the Fedora Directory Server (`http://directory.fedoraproject.org/`). We haven't used it, but the existence of a graphical utility for Fedora Directory Server administration will surely help many new LDAP administrators.

We think LDAP is a great system for a medium to large company or other organization. It takes a bit of work to set up, and you have to make sure your systems can take advantage of it, but it is worth it when you have a lot of account information to manage. If you decide to use LDAP, take a look at *LDAP System Administration* by Gerald Carter (O'Reilly Media Inc., 2003).

Security with Kerberos

Kerberos is an authentication system designed to be used between trusted hosts on an untrusted network. Most commonly, a Kerberos server is used to authenticate remote users without sending their passwords over the network. Kerberos is a pretty common security system and basic information can be found at `http://web.mit.edu/kerberos/www/`.

Kerberos is the best option (that we know of) available today for authenticating the same accounts across multiple systems. Unlike many other options, the users' passwords are rarely sent over the network. When they are, they are strongly encrypted.

Using Kerberos for authentication on your systems is not always easy, unfortunately. First of all, you need to set up a Kerberos server, which is beyond the scope of this book. It isn't the hardest thing in the world to do, but it will require a fairly serious time investment. Good documentation can be found at MIT's Kerberos site: `http://web.mit.edu/kerberos/www/`.

You will also need to make sure any programs that require user authentication on your systems are able to use Kerberos. Most systems support PAM, which allows you to use Kerberos easily for all system-level authentication. If you do have PAM, probably most of the applications that came with your systems and require authentication can also use PAM. Other applications, like Apache and Samba, may directly support Kerberos as well (with or without PAM).

Another advantage of Kerberos is its ability to use one authentication service from several unique software packages. It is not uncommon for each user to have a separate password for logging into systems over SSH, accessing a restricted web server, and accessing a Samba share. With Kerberos, you can use the same user password for all of these different services and any other services that support Kerberos.

Like LDAP, Kerberos is an excellent choice if you have a large number of user accounts and a decent number of systems. In fact, if you have a large enough number of systems, it can be worth the effort regardless of the number of accounts you use. Because Kerberos is also the safest way to authenticate users over the network and can be used from such a wide variety of software, it is something you should consider using in almost any environment.

Implementing Host-Based Firewalls

Firewalls are any hardware or software that blocks or otherwise disallows IP traffic, based on rules or policy settings. Deploying firewalls at the periphery of a network, usually on or near the links that connect to other networks or to the Internet, is common practice. In recent years, it has become increasingly common for individual computers to run firewall software.

Even if a host isn't running any unneeded network daemons, a local firewall can help in several ways:

- If unwanted traffic makes it through a perimeter firewall, a local firewall can still block it. The practice of running several, redundant security systems at once is called defense in depth and is a wise way to handle security.

- A system can prevent connections from unwanted hosts on the local network where there is no network-based firewall between the hosts.

- UNIX operating systems sometimes require daemons to run and listen on the network in order for the base system to work properly. There may be no need for the daemon to accept connections from remote hosts, so protecting the program with a firewall is the only remaining option for protecting this service from the network. This problem is less prevalent with base UNIX installs these days, but this issue might still come up with third-party software.

Software that blocks IP traffic directly in a system's TCP/IP stack is called packet-filtering software. True to their name, packet filters use attributes of an incoming packet such as source IP and destination port to block and/or allow network traffic.

Software that proxies connections and only allows permitted application protocol operations is also a firewall, but we don't cover proxying in this book. We *do* recommend that you evaluate the use of proxy software for both inbound and outbound traffic at your site, where appropriate.

Software that runs outside the operating system kernel to block traffic is also firewall software, though most people don't think of it as such. Software such as TCP Wrappers (covered in the next section) fits this description.

Using TCP Wrappers

You will always want some network services to remain active. If any of these services are executed by inetd, using TCP Wrappers is a good idea. TCP Wrappers is a program (usually named tcpd or in.tcpd) that can be executed by inetd. It performs some checks on the network connection, applies any access control rules, and ultimately launches the necessary program.

All of the systems in our example network come with TCP Wrappers installed by default.

Even though the TCP Wrappers program is already installed (in a location like /usr/sbin/tcpd), you need to make sure your systems use it. A system without TCP Wrappers enabled would have a /etc/inetd.conf with entries like this (your file location and entry format may vary):

```
ftp stream tcp nowait root /usr/sbin/in.ftpd in.ftpd
telnet stream tcp nowait root /usr/sbin/in.telnetd in.telnetd
```

To activate TCP Wrappers, you want to modify these entries to call the `tcpd` program as follows:

```
ftp stream tcp nowait root /usr/sbin/tcpd in.ftpd
telnet stream tcp nowait root /usr/sbin/tcpd in.telnetd
```

You can do this using the `editfiles` section:

```
editfiles:
  { /etc/inetd.conf
    ReplaceAll "/usr/sbin/in.ftpd" With "/usr/sbin/tcpd"
    ReplaceAll "/usr/sbin/in.telnetd" With "/usr/sbin/tcpd"
    DefineClasses "modified_inetd"
  }
```

This will cause your system to use TCP Wrappers for both the FTP and Telnet services.

■**Note** We don't recommend using Telnet for remote system access; this is for demonstration purposes only.

Don't forget to send the `HUP` signal to `inetd`:

```
processes:
  modified_inetd::
    "inetd" signal=hup
```

Simply enabling TCP Wrappers enhances the security of the selected network services. You can gain additional benefits by restricting access to these services using `/etc/hosts.allow` and `/etc/hosts.deny`. A properly configured corporate firewall, a system-level firewall if possible (as described in the next section), and TCP Wrappers with access control enabled provide three tiers of protection for your network services. Using all three may seem like overkill, but when you can do all of this automatically, there really is little reason not to be overly cautious. Any one of these security devices could fail or be misconfigured, but probably not all three.

Using Host-Based Packet Filtering

As previously mentioned, packet filtering is a way of allowing or disallowing IP traffic as it comes into a system's network interface, based on filtering rules. All three of our example

operating systems at our site install packet-filtering software with the base system. On both Linux distributions (Debian and Red Hat), the iptables software is used, and on Solaris the ipfilter software is used.

In this section, we'll provide a quick introduction to iptables and demonstrate how to fully enforce a local host packet filtering policy. From there, it will be up to you to configure a firewall policy that's appropriate for your site.

For help with ipfilter, consult the project home page at `http://coombs.anu.edu.au/ ~avalon/` and the Sun online documentation at `http://docs.sun.com/app/docs/doc/816-4554/ eupsq?a=view`.

Iptables on Debian

Iptables is the packet filtering framework used by the Linux kernel since major version 2.4. It consists of kernel code and user-space tools to set up, maintain, and inspect the kernel tables of IP packet filter rules. Each table contains several built-in chains and may also contain user-defined chains.

A chain is simply a list of iptables rules with patterns to match particular packets. Each rule specifies a target, which defines what to do with the packet (i.e., allow or drop the packet). A target can also be a jump to a user-defined chain in the same table.

Our Red Hat systems have an iptables firewall installed and configured at boot, as automated by our Kickstart configuration. We also automated the distribution of the firewall configuration file to our Red Hat web server (using cfengine) back in Chapter 10, so that we could remotely connect to the NRPE daemon. Since iptables on Red Hat is already configured and automated on our network, we'll focus on setting up packet filtering for Debian. We'll focus on our Debian-based log host, called *loghost1*. This host is ideal because of its security-related duties.

In order to set up iptables on Debian, we'll need to

1. Define a firewall policy

2. Create iptables rules that implement our policy.

3. Copy the file to *loghost1* using cfengine.

4. Configure the system to start the firewall rules before the network interfaces are brought up.

5. Restart a network interface or reboot the host, and verify our firewall settings.

We think a very simple firewall policy is appropriate for our log host. We will allow incoming network connections only for these daemons and disallow the rest:

- syslog-ng

- NRPE

- sshd

- cfengine (to the `cfservd` daemon)

The daemons and processes on the local system that connect to services on remote hosts will be allowed by our policy, as well as any return traffic for those connections. Any incoming traffic to services other than those listed previously will be blocked.

The rules defined by iptables are enabled by the `iptables` command line utility. Rules apply to traffic as it comes into an interface, as it leaves an interface, or as it is forwarded between interfaces.

An iptables rule set that implements our log host policy follows. The rules are evaluated in order, and packets that fail to match any explicit rules will have the default policy applied.

```
#!/bin/sh

# make sure we use the right iptables command
PATH=/sbin:$PATH

# policies (policy can be either ACCEPT or DROP)
# block incoming traffic by default
iptables -P INPUT DENY
# don't forward any traffic
iptables -P FORWARD DENY
# we allow all outbound traffic
iptables -P OUTPUT ACCEPT

# flush old rules so that we start with a blank slate
iptables -F

# flush the nat table so that we start with a blank slate
iptables -F -t nat

# delete any user-defined chains, again, blank slate :)
iptables -X

# allow all loopback interface traffic
iptables -I INPUT -i lo -j ACCEPT
```

```
# A TCP connection is initiated with the SYN flag.

# allow new SSH connections.
iptables -A INPUT -i eth0 -p TCP --dport 22 --syn -j ACCEPT
# allow new cfengine connections
iptables -A INPUT -i eth0 -p TCP --dport 5308 --syn -j ACCEPT
# allow new NRPE connections
iptables -A INPUT -i eth0 -p TCP --dport 5666 --syn -j ACCEPT
# allow new syslog-ng over TCP connections
iptables -A INPUT -i eth0 -p TCP --dport 51400 --syn -j ACCEPT

# allow syslog, UDP port 514. UDP lacks state so allow all.
iptables -A INPUT -i eth0 -p UDP --dport 514 -j ACCEPT

# drop invalid packets (not associated with any connection)
# and any new connections
iptables -A INPUT -m state --state NEW,INVALID -j DROP

# stateful filter, allow all traffic to previously allowed connections
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

# no final rule, so the default policies apply
```

Note that this is a shell script. It is possible to save currently active iptables rules using the `iptables-save` command, by redirecting the command's output to a file and loading the rules via the `iptables-restore` command. We configure our host using a shell script because this setup is easy for you use and experiment with on your own.

We copied this script to into the directory /etc/network/if-pre-up.d/ on *loghost1* and made sure the script was executable. Scripts in this directory are run *before* the network interfaces are brought up. Linux allows firewall rules to be defined for interfaces that don't exist, so with this configuration we never bring up interfaces without packet filtering rules.

Note that we didn't go into the details of how we copied the file using cfengine. By this point in the book, we think that you are probably an expert at copying files using cfengine and don't need yet another example.

Once the iptables script was copied in place, we rebooted the host *loghost1*. When it came back up, we ran this command as root to inspect the current iptables rule set:

```
# iptables -L -n -v
Chain INPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target    prot opt in out source destination
 11  1084 ACCEPT udp -- eth0 * 0.0.0.0/0 0.0.0.0/0 udp dpt:514
  5   300 ACCEPT   tcp -- eth0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:51400 flags:0x17/0x02
  0     0 ACCEPT    tcp -- eth0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:5666 flags:0x17/0x02
  0     0 ACCEPT    tcp -- eth0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:5308 flags:0x17/0x02
  3   180 ACCEPT   tcp -- eth0 * 0.0.0.0/0 0.0.0.0/0 tcp dpt:22 flags:0x17/0x02
 328 67720 ACCEPT 0 -- lo   *  0.0.0.0/0 0.0.0.0/0
 76  8793 DROP     0 -- *   *  0.0.0.0/0 0.0.0.0/0  state INVALID,NEW
 1033  157K ACCEPT  0 -- * *  0.0.0.0/0 0.0.0.0/0  state RELATED,ESTABLISHED

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target    prot opt in    out     source             destination

Chain OUTPUT (policy ACCEPT 1712 packets, 216K bytes)
 pkts bytes target    prot opt in    out     source             destination
```

We ran `iptables` with the options to list all rules in all chains and to disable DNS resolution for the IP addresses listed in the rules. The long lines wrap around the page, making it harder to read, but the active rule set matches our policy. The host *loghost1* will not allow any inbound network traffic except that required for administration, monitoring, and the one network service it offers to the network: syslog.

For further information on iptables consult the iptables home page at `http://www.netfilter.org/`.

Enabling Sudo at Our Example Site

We discussed `sudo` extensively in Chapter 1, but we didn't have any systems to deploy it to back then. We need to start using `sudo` at our example site.

The `sudoers` file installed by the `sudo` package on Red Hat has a rich set of commands grouped into related tasks, which can be easily delegated to users with particular roles. You might find it to be a good starting point for the global `sudoers` file at your site.

We used the default Red Hat `sudoers` file for a new file at the location `PROD/repl/root/etc/sudoers/sudoers` (plus we added it to our Subversion repository), and simply added this line:

```
%root           ALL=(ALL)       ALL
```

To have a good audit trail, we want administrators to execute commands that require root privileges with single sudo command like this:

```
$ sudo chmod 700 /root
```

This way, root commands are logged via syslog by the sudo command, so our log host gets the logs, and the regular logcheck reports will include all commands run as root.

There is a problem, though. Nothing stops our administrators from running a command that gives them a root shell:

```
$ sudo /bin/sh
```

When a shell is executed as root, sudo will not (and cannot) log each command run inside the shell. This is a blind spot in our audit trail, and the way to avoid this is to not give unlimited sudo command access to administrators. Instead, you should build on the examples provided in the Red Hat sudoers file to provide only the needed set of commands to your administrator staff. Here are some example entries from the Red Hat sudoers file that delegate privileges in a desirable manner (slightly modified for example purposes):

```
## User Aliases
## These aren't often necessary, as you can use regular groups
## (ie, from files, LDAP, NIS, etc) in this file - just use %groupname
## rather than USERALIAS
User_Alias ADMINS = nate, kirk

## Command Aliases
## These are groups of related commands...

## Networking
Cmnd_Alias NETWORKING = /sbin/route, /sbin/ifconfig, /bin/ping, /sbin/dhclient,
 /usr/bin/net, /sbin/iptables, /usr/bin/rfcomm, /usr/bin/wvdial, /sbin/iwconfig,
 /sbin/mii-tool

## Installation and management of software
Cmnd_Alias SOFTWARE = /bin/rpm, /usr/bin/up2date, /usr/bin/yum

## Next comes the main part: which users can run what software on
## which machines (the sudoers file can be shared between multiple
## systems).
## Syntax:
```

```
##
##      user    MACHINE=COMMANDS
##
## The COMMANDS section may have other options added to it.

## Allows members of the 'sys' group to run networking
%sys ALL = NETWORKING

## Allows the ADMINS user alias to run software commands
ADMINS ALL = SOFTWARE
```

The two command aliases (`SOFTWARE` and `NETWORKING`) are perfect examples of using roles to delegate privileges. If a user or administrator needs access to only commands to modify network settings or to install software, the preceding command aliases allow this. The delegation of `NETWORKING` to a group of users is done via traditional UNIX group membership in this example, and the delegation of `SOFTWARE` privileges is done via a list of users in the `sudoers` file itself.

■Note Always check to make sure the commands you enable, especially the ones that grant root privileges, don't have shell escapes. Shell escapes are features that allow shell commands to be executed. Any such commands will run with root privileges and completely circumvent the access limitations that we're using `sudo` for in the first place.

To copy our new `sudoers` file to the hosts in our example environment, we added a task to `PROD/inputs/tasks/app/sudo/cf.copy_sudoers` with these contents:

```
control:
        any::
                AllowRedefinitionOf     = ( sudoers_destination )
                sudoers_destination     = ( "/etc/sudoers" )

        solaris|solarisx86::
                sudoers_destination     = ( "/opt/csw/etc/sudoers" )

copy:
        any::
                $(master_etc)/sudoers/sudoers
                        dest=$(sudoers_destination)
                        mode=440
```

```
                          owner=root
                          group=root
                          server=$(fileserver)
                          type=checksum
                          encrypt=true
                          inform=true
```

We imported the task in `PROD/inputs/hostgroups/cf.any`, committed our change to Subversion, and checked it out to the live `PROD` tree on the cfengine master.

We were dismayed to find that we are missing the `sudo` package on our Debian hosts. To get `sudo` installed via FAI on future Debian installs, we added the line `sudo` to `PROD/repl/root/srv/fai/config/package_config/FAIBASE`. Our Solaris Jumpstart postinstall script already installs `sudo`, and our Red Hat systems come with it as well. For now, you can manually install `sudo` on your Debian systems to avoid reimaging just for one package to get installed.

Be sure to add `ignore=/usr/bin/sudo` to the Debian section of the `PROD/inputs/tasks/os/cf.suid_removal` task so that `sudo` actually works for more than one day!

Security Is a Journey, Not a Destination

We need to be mindful that a secure state is never reached. We can only increase security to where we feel that we have decreased the risk of successful penetration to a low level. We need to keep up to date with security announcements and have security in mind with all administrative activities at our site.

We have now enhanced the security at our site by reducing the overall exposure of our systems to the network, as well as to local threats. Even if an attacker gained access to one of our systems using a nonprivileged account, only a limited number of SUID binaries owned by root can be used for privilege escalation, and local software should be up to date and therefore free of publicly known vulnerabilities.

Host-based security measures are the final line of defense when network firewalls fail to protect our internal hosts. Systems that run daemons that are accessible from outside networks (or the Internet at large) should also be firewalled off from internal networks such as workstation and internal server networks. These measures help prevent exposure in the event that a remote attacker gains access to systems that have to be exposed to hostile networks themselves.

The final weak spot that we didn't cover is that of any internally developed software in use at your site. Such software is especially risky if it is exposed to other networks or the Internet at large. Security advisories and vendor announcements address problems with vendor and open source software, but only source code audits by trusted third parties and good coding practices can protect internally developed software. If you support such software, be sure to take extra steps to firewall the hosts running the software from the rest of the hosts on your network.

APPENDIX A

■ ■ ■

Introducing the Basic Tools

Because this book is written for an experienced administrator, we have made a good number of assumptions about your knowledge and previous experience. In an effort to provide a helping hand to some of you without boring others, we are providing a basic introduction to several of the tools we have used throughout this book in this appendix.

This appendix only provides an introduction and usage examples for the basic tools and technologies used throughout the main text. Many other tools are described in detail as they are introduced in the chapters, and thus they won't be discussed here. In addition, tools like cfengine are covered in numerous chapters as appropriate.

In this appendix, we provide enough information so that if you are unfamiliar with the topic, you should still be able to understand the examples within this book. Hopefully, this appendix will also give you enough information to start exploring the tools on your own. If you are not ready to do that, just refer to the additional resources we provide in most sections.

Throughout this book, we have not attempted to cover every existing utility for each task. We generally pick the most popular tool from each category. If a couple of the options are different in scope or power, we try to cover both of them. The same is true for this appendix: we talk about the most popular choice and try to mention other tools that you could also use.

For each section in this appendix, we recommend reading the first few paragraphs to get a feel for how that tool or technology is used in this book. At that point, if you are already familiar with the topic being covered, you can skip to the next section.

The Bash Shell

The Bourne-Again Shell (Bash) seems to be the most common command interpreter in use today. Many people use it for their interactive sessions as well as for creating shell scripts.

We highly recommend that you basically understand shell scripting in at least one shell before you read this book. If you are fairly good with another shell, you should do fine with the Bash examples. If you have no experience with shell scripting, we would recommend that you become familiar with the Bash shell before you continue. This section

provides a basic introduction to Bash and mentions some resources for further reference. Of course, nothing is better than using the shell and experimenting with scripts yourself.

Compatibility Issues with Bash

Bash executes scripts written for the original Bourne Shell (sh) in addition to native Bash scripts. In fact, on some systems, sh is actually a symbolic link to Bash. When run through this symbolic link, Bash executes in a compatibility mode so that it operates similarly to the original sh.

Of course, Bash has quite a number of features not available in the traditional Bourne Shell. If you use any of these additional features, your script will no longer be compatible with the original Bourne Shell. Even so, most of the Bash scripts used throughout this book will work fine with the older Bourne Shell with few if any modifications.

If your environment contains systems that have only the Bourne Shell installed, you need to keep that in mind when you write your own shell scripts. In most cases, however, it is worth the effort to install Bash (or the shell of your choice) on every machine you are administering. You will almost certainly need a method of installing software on all of your systems, so installing a consistent shell across your systems is a good place to start.

■**Caution** Some systems, such as Solaris, always use /bin/sh when executing system initialization scripts. Since replacing /bin/sh with Bash is not really a good idea, you probably want to write these types of script in sh format so that they will operate on any system.

Creating Simple Bash Shell Scripts

You will primarily use the Bash shell as a command interpreter. Many people choose this shell for their command-line interpreter because of features like command history, tab-completion, and user-defined functions. In this book, however, we focus on its scripting abilities.

In their simplest form, shell scripts are simply text files with a sequence of commands to execute. Any command that can be executed at the command prompt can be placed into a shell script. Here is a sample shell script that illustrates some basic activities with bash:

```
#!/bin/bash

# Ask the user for a directory
echo "Please enter a directory name:"
read input
```

```
# Now, show the contents of that directory
echo
echo "Contents of directory $input:"
ls $input
```

■Note You can find the code samples for this appendix in the Downloads section of the Apress web site
(http://www.apress.com).

The first line, like any interpreted script, contains the characters #! followed by the
full path to the interpreter (in this case, /bin/bash). The rest of the script contains shell
commands, all of which could be typed directly at the command prompt that executes
when the script runs. You run the script like any other binary program, by making it exe-
cutable and then executing it. Assuming the file is named bash_example.sh, you would run
it as follows:

```
$ chmod u+x bash_example.sh
$ ./bash_example.sh
Please enter a directory name:
/usr

Contents of directory /usr:
bin dict etc i386-glibc21-linux kerberos libexec man share tmp
coda doc games include lib local sbin src X11R6
```

Any command that you can run from the command line can be run in a shell script.
The shell also provides all of the traditional logic structures (if, then, else blocks; while
loops; etc.). Since there are so many example shell scripts in this book, and since we
explain the examples as we go along, we will not try to provide a comprehensive intro-
duction to them here.

Debugging Bash Scripts

If you have trouble understanding some of the scripts in the book, try simply running
the script yourself. Like in any other language, you can insert echo commands into exist-
ing scripts to see the values of variables at various points. You can also use the -x option
that Bash provides to see exactly what commands are running. Here is the example script
being run with that option:

```
$ bash -x bash_example.sh
+ echo 'Please enter a directory name:'
Please enter a directory name:
+ read input
/usr
+ echo

+ echo 'Contents of directory /usr:'
Contents of directory /usr:
+ ls /usr
bin doc games kerberos libexec man share tmp
dict etc include lib local sbin src X11R6
```

Note that the lines starting with + shows you the command that is about to run, right before it runs. For a complex script, you may want to capture all of the output to a file and then look through the results in a text editor:

```
$ bash -x somescript.sh >log.out 2>&1
```

You can also modify the line at the top of the script (`#!/bin/bash`) to always enable debugging: `#!/bin/bash -x`. You can even enable debugging at any time during a script's execution by using the `set -x` command.

The `-x` option is only one of a variety of debugging options available to you. All of these can be specified on the command line, on the interpreter line, or using the `set` command:

- `-n`: This switch causes the script to be parsed and checked for syntax errors. No commands are actually executed.

- `-v`: This displays every line of code before execution, including comments. It is similar to the `-x` switch, but in this case, the lines are printed before any parsing is performed on them.

- `-u`: This one causes an error message to be generated when an undefined variable is used.

Other Shells

Many shells are available in addition to Bash. Although we think that we can safely say that Bash has become the most common shell, choosing one shell as the "best" would be a difficult task. In fact, even attempting such a feat would cause an immediate religious war. So, the best we can do is list a few other popular shells and let you do your own investigating if you so desire. Each shell has a different syntax for its scripts, but they also have many similarities. Here are other popular shells that you may want to investigate:

- C shell (`csh`)

- Korn shell (`ksh`)

- `zsh`

Bash Resources

The (large) man page for Bash contains a lot of information and can be useful for reference (you access it by running `man bash`). Usually more helpful is the actual `help` command that provides information on all of the built-in Bash commands, including control constructs often used in shell scripts, for example:

```
$ help while
while: while COMMANDS; do COMMANDS; done
    Expand and execute COMMANDS as long as the final command in the
    'while' COMMANDS has an exit status of zero.
```

Finally, if you have never used the Bash shell before and you want to improve your skills with the shell, or if you just want a nice reference, *Learning the bash Shell*, by Cameron Newham and Bill Rosenblatt (O'Reilly Media Inc., 2005), is a great book on the topic.

Perl

Perl is very popular in the system administration community and may very well be the most popular scripting language in use today. The major disadvantage is that some commercial UNIX variants may not come with Perl as standard software. However, most administrators find that adding Perl to all of their UNIX systems is well worth the effort, and it comes preinstalled on all major Linux distributions.

The advantages of Perl are plentiful. It is an extremely powerful scripting language and significantly faster than shell scripts. It is (in our experience, at least) very reliable and stable. You can find existing Perl scripts that can perform a wide variety of tasks—these make great examples and starting places for your own scripts.

Perhaps the best resource for Perl is the Comprehensive Perl Archive Network (CPAN), which contains huge numbers of modules that can add almost any functionality to the language. You can find these modules at `http://www.cpan.org` or `ftp://ftp.cpan.org`. These modules were contributed by thousands of Perl developers and systems administrators around the world. They can save you a lot of time, and if you upload your own modules, you can save other people time as well.

The major complaint people have about Perl is that the source code is hard to read. Part of the reason for this is that there are two camps in program language design: one camp thinks the programming language should force programmers to write readable

and maintainable code; the other camp thinks programmers should be able to write code any way they want to. Larry Wall (the main author of Perl) is in the latter camp. What this means is that you can write Perl code any way you like—messy or clean. We, of course, recommend that you write clean, clear, and well-documented code, and we attempt to do so for all of the Perl examples within this book.

Basic Usage

The following is a very simple (and useless) program that lists the contents of a directory using both the system's ls command and internal Perl functions. It does illustrate some basics about Perl, though.

```perl
#!/usr/bin/perl -w
use strict;

# First, ask user for a directory
print "Please enter a directory name: ";

# Use 'my' to declare variables
# Use the <> operators to read a line from STDIN
my $dir = <STDIN>;

# Now, use the 'chomp' function to remove the carriage return
chomp($dir);

# First, do a dir listing by executing the 'ls' command directly
print "\nCalling system ls function:\n";
# The 'system' function will execute any shell command
system("ls $dir");

# Next, do a dir listing using internal Perl commands
# The 'die' function will cause the script to abort if the
# 'opendir' function call fails
print "\nListing directory internally:\n";
opendir(DIR, $dir) or die "Could not find directory: $dir\n";
my $entry;
# Now, read each entry, one at a time, into the $entry variable
while ($entry = readdir(DIR)) {
   print "$entry\n";
}
closedir(DIR);
```

Just like in a shell script, the first line must contain the path to the Perl interpreter. On most Linux machines, this will be `/usr/bin/perl`. On many UNIX machines, it will be `/usr/local/bin/perl`.

One thing to note is our use of the `-w` option to the interpreter. This, combined with the second line of the script (`use strict`), causes Perl to require variables to be declared before they are used and to provide useful compilation warnings and other valuable information. It is considered good practice to use these settings for all Perl programs to help avoid errors and aid in the debugging process.

The example script should be generally self explanatory. Here is the script being executed:

```
Please enter a directory name:

/tmp/test
Calling system ls function:
file1  file2  file3

Listing directory internally:
.
..
file1
file2
file3
```

Notice that this version of `ls` did not hide the hidden directories `.` and `..` (the single dot and the double dots). It also did not do the listing in a space-efficient multicolumn format. You could easily enhance this Perl script in this manner if you so desired. Providing all of the capabilities of the system `ls` command would, however, be more difficult simply because it has such a wide variety of command-line options.

Like with the shell scripts in the previous section, our discussion of Perl cannot be comprehensive. Hopefully, the examples and the accompanying explanations throughout this book will be enough for you to gain a basic knowledge of Perl. If you have problems, be sure to use the documentation provided with Perl and/or the great Perl books available.

When using other people's Perl programs, you may find that they require certain Perl modules that you do not have installed on your system. You can find these modules at `http://www.cpan.org`. You can also try using the `CPAN` module to automatically install other modules for you. You can do this by running the following command as `root`:

```
# perl -MCPAN -e 'install Some::Perl::Module'
```

There is one other complication you will always have with Perl scripts. If you download a Perl script from somewhere, the first line is always the path to the Perl interpreter,

but it may not be the path to *your* Perl interpreter. You will see all kinds of paths in the scripts that you download: `/usr/bin/perl`, `/usr/local/bin/perl`, `/opt/perl/bin/perl`, and even `/export/homes/home1/joe/programs/development/languages/perl/bin/perl`.

Likewise, if you have a mix of systems, the path to your Perl interpreter might not be fixed. Your Linux systems might have Perl in `/usr/bin/`, but the rest of your systems might have it in `/usr/local/bin/perl`. You will save yourself a lot of time by standardizing the location of Perl across your systems—create some symbolic links if necessary.

Now, all you have to do is make sure all of your Perl scripts are using the path to the interpreter that is valid for your systems. Here is a simple shell script that takes care of this for you for all files you provide as arguments:

```
#!/bin/bash

for file ; do
    sed '1 s=^#![^ ]\+perl=/usr/bin/perl=' "$file" > "$file.new" \
        && mv "$file.new" "$file"
done
```

You should obviously replace the string `/usr/bin/perl` with the proper path for your Perl interpreter.

Other Scripting Languages

Although system administration tools could be written in traditional languages such as C, scripting languages are generally used. Scripting languages are nice, because they can be distributed in their original text and will work on any supported platform. Calling other shell utilities is also much easier with scripting languages than with compiled languages. And, as you will find, plenty of shell utilities will make your life much easier.

Although Perl is used quite a bit within this book, several other scripting languages can do many of the same tasks just as well. The most popular include these:

- *Python*: Python is a language that has gained a lot of popularity with system administrators. Python programs tend to be more structured than Perl programs, so Python may be a better choice for more complicated programs as well as to ensure easy understanding and debugging by other administrators.

- *Tcl*: Tcl is a relatively old language that is especially popular for providing GUI interfaces. You can use the Expect program, which is a Tcl extension, to automate interaction with programs that are designed to be interactive.

- *AWK*: AWK is not as powerful as Perl, Python, or Tcl for many tasks, but for text processing, it can be very convenient and powerful. GNU's extended version of AWK, gawk, is also a popular text editing tool.

Perl Resources

The de facto Perl book is *Programming Perl*, by Larry Wall and others (O'Reilly Media Inc., 2000). This book provides a great introduction as well as plenty of details on the language. Even if you already know Perl, *Perl for System Administration*, by David N. Blank-Edelman (O'Reilly Media Inc., 2000) would be a great companion to this book.

In addition to these books, Perl comes with quite a bit of documentation. For starters, there is the `perl` man page (which refers you to additional man pages). Perl also comes with a convenient `perldoc` command. `perldoc File::Copy` provides documentation on the `File::Copy` Perl module. `perldoc -f system` provides help on the built-in system function. Finally, `perldoc -q term` searches the FAQ for the given term.

Basic Regular Expressions

On the command prompt, you can type a command like `rm a*`. The `a*` is expanded (by the shell) to all file names beginning with the letter `a`. This is called file globbing.

Regular expressions are very similar in concept, but they have many differences. Instead of working with files, they work with text, usually on a per-line basis. They also have a wider variety of operators than file globbing has.

There are many different implementations of regular expressions, which can sometimes lead to confusion. Some of the common programs that use regular expressions yet have at least some differences in their implementations are `grep`, `egrep`, AWK, gawk, `sed`, and Perl. In this section, we present the basics of using regular expressions that are commonly found in most regular expression implementations. You will need to check the documentation for each specific program to find out about its nuances.

Characters

The most basic representation in a regular expression is that of a character. Most characters represent themselves—the character `a` matches the letter `a`, for example. Other special characters need to be escaped with a backslash to represent themselves. To match the character `[`, for example, you need to write `\[`.

You might be tempted to backslash all nonalphanumeric characters just to be safe. In some implementations (like Perl), this works pretty well. In other implementations (like `sed`), unfortunately, this approach can backfire. You must use caution when you use certain characters in a new program. These often have a special meaning by themselves in some implementations and when they are escaped in other languages. These include `(`, `)`, `{`, `}`, and `+`.

The period character (`.`) is a special character that matches any single character (just like the `?` in file globbing). The regular expression `lake` will match the word "lake". The

regular expression `.ake` will also match the word "lake" in addition to the words "make" and "take". To match a literal period character, you must use `\.`.

You can also use character classes, which allow you to match a selection of characters. The sequence `[abc]` matches any single character: a, b, or c. You can create inverse character lists by placing the special character ^ first in the list: `[^abc]`. This matches any single character that is not a, b, or c.

There is a common shortcut for placing many characters in a character class. The sequences `[0123456789]` and `[0-9]` both match any single numerical character. The sequence `[a-zA-Z0-9]` matches any single alphanumeric character.

Many implementations have other shortcuts available. For example, you can use `[[:digit:]]` to match any digit in `egrep` and `\d` to match any digit in Perl. Most implementations have several classes of characters that can be represented in this manner.

Matching Repeating Characters

You can use available tools to match sequences of characters. All of these must be preceded by a single character or character class that they allow to be matched multiple times:

- `?`: Match zero or one of the character(s).

- `*`: Match the character(s) zero or more times.

- `+`: The character(s) must match one or more times. Note that this is not supported in all implementations. The `sed` command does not traditionally recognize this repetition operator, but the GNU version supports the `\+` operator with the same results.

You will often find these characters preceded by a `.`. The sequence `.*`, for example, will match zero or more of any character (just like * in file globbing).

There are lots of other possibilities. The sequence `[abc]+` will match one or more of the characters a, b, or c. It will match the strings `abc` and `aabbcc`. It will also match portions of the strings `dad` and `zbbbz`. It will not match the string `d`, however, because at least one match must be found.

You can find a few additional repetition operators in some implementations of regular expressions:

- `{x}`: The character(s) must be matched exactly x times.

- `{x,}`: The character(s) must be matched at least x times.

- `{x,y}`: The character(s) must be matched at least x times but not more than y times.

So, the sequence a{2} will match the string aa but not a. These operators are not present in some implementations. In others, the curly braces must be backslashed (a\{2\}). Note that the sequence {,y} (i.e., no more than y times) does not usually work.

Other Special Characters

A few additional characters have special meanings:

- ^: Match the beginning of a line or the beginning of the buffer.

- $: Match the end of a line or the end of the buffer.

- |: Join the expressions on the left and right with a logical OR.

So, given this information, you can see that the regular expression mad will match "mad", "made", and "nomad". The regular expression ^mad$, however, will match only "mad".

You can use the | character to join two regular expressions together, allowing one or the other to be matched. In some implementations (like sed), it must be backslashed. This allows you to two different words (such as hello|bye).

Sometimes, you may want to use parentheses to group the |operator. The expression ^a+|b+c+$ matches either a string of all as or a string with any number of bs followed by any number of cs. The expression ^(a+|b+)c+$, on the other hand, only matches strings ending in cs but beginning with either as or bs. In some implementations, the parentheses might need to be backslashed when used as grouping operators.

Marking and Back Referencing

Parentheses (or backslashed parentheses in implementations such as sed) mark sequences in addition to their grouping functionality. These marked portions of the string being searched can be referenced later in your regular expression.

Each marked string is assigned the next number in a series, starting with 1. If the regular expression (.)(.)(.*) is applied to the string abcdefg, for example, \1 would contain a, \2 would contain b, and \3 would contain cdefg.

You can also nest parentheses, in which case the outermost set of parentheses come first. So when the regular expression (a(b)) is applied against the string ab, \1 will contain ab and \2 will contain b.

In most languages, you refer to a back reference with the sequence \x, where x is the number of the marked string you want to reference. The regular expression ([a-zA-Z]+)-\1, for example, will match any string that contains two identical words separated by a hyphen; it will match "dog-dog" but will not match "cat-dog".

Back references are most commonly used when you are using a regular expression to make modifications (like with sed) or to retrieve information from a string (like with Perl). In sed, the first marked string is \1 and the entire matched string is \0. In Perl the first marked string is $1 and the entire matched string is $0. Here are a couple of quick examples with sed (for more information on sed, see "The sed Stream Editor" later in this appendix):

```
% echo abcdef | sed 's/\(ab*\)c\(.*\)/\1 \2/'
ab def
% echo abbcdef | sed 's/\(ab*\).*\(.\)/\1 \2/'
abb f
```

The second example illustrates one last concept—greediness. The b* sequence matched as many characters as it could, so it matched both b characters. The following .* could also have matched both b characters, but the b* came first in the regular expression. The .*, on the other hand, could have matched all the way to the end of the expression, including the f. If this would have happened, though, the entire expression would have failed, because the final . would have nothing left to match. For this reason, the .* matched as many characters as it could while still allowing the entire expression to be successful.

In some implementations, like Perl, a repetition operator can be followed by a ? to make it nongreedy, which causes the repetition operator to match as few characters as possible.

grep

grep is a very old program that looks for the specified search string in each line of input. Every line that it matches is displayed on stdout. It can also take basic regular expressions. You can find grep on just about any UNIX system.

The egrep command is a newer version of grep that supports extended regular expressions (such as the + repetition operator). Some implementations even support the {} repetition operators (and others support \{\} instead). The egrep command can also be found on many systems.

If you find yourself limited by the standard grep command and the differences between the various egrep implementations, consider installing a standard version (such as GNU egrep) on all of your systems. If your script is designed to run on your own systems, this is a reasonable solution. If your script is designed to run on any arbitrary system, you will have to stick with the lowest common denominator.

Many of the following examples will use this sample input file, called input_file:

```
line 1
hello, I'm line 2
this is line 3
```

Let's start out with a simple example:

```
$ cat input_file | grep 'hello'
hello, I'm line 2
```

The `grep` command filtered the input file and displayed only the lines matching the regular expression (or just a string in this case) `hello`. Here are two more ways the same result could have been obtained:

```
$ grep 'hello' <input_file
hello, I'm line 2
$ grep 'hello' input_file
hello, I'm line 2
```

You can even list multiple files on the command line—as long as your regular expression comes first. Here is a regular expression being processed by the `egrep` command (we must use `egrep` because `grep` does not recognize the + operator):

```
$ egrep '^.+line [0-9]$' input_file
hello, I'm line 2
this is line 3
```

Here, we matched only lines that contained text before the `line X` string (where `X` is a single digit from `0` to `9`). We could also have used the `-v` switch to invert the output (i.e., display nonmatched lines) and used a simpler regular expression:

```
$ grep -v '^line' input_file
hello, I'm line 2
this is line 3
```

Within scripts, using `grep` to simply check for the presence of a line is common. The `-q` switch tells `grep` to hide all output but to indicate whether the pattern was found. An exit code of `0` (true) indicates the pattern was found on at least one line. An exit code of `1` means the pattern was not found on any line. Here are two examples:

```
$ grep -q 'foo' input_file && echo 'Found'
$ grep -q 'line' input_file && echo 'Found'
Found
```

You can also have grep indicate the number of lines that were matched:

```
$ grep -c 'line' input_file
3
```

One common command-line use of grep is to filter output from system commands. This is often handy within shell scripts as well. To see only the processes being run by the user kirk, for example, you can try this:

```
$ ps aux | grep '^kirk'
kirk      1103  0.0  0.2  4180 1040 pts/0     S    09:41   0:00 bash
kirk      1109  0.0  0.2  4180 1040 pts/1     S    09:41   0:00 bash
kirk      1110  0.0  0.2  4180 1040 pts/2     S    09:41   0:00 bash
kirk      1113  0.0  0.2  4180 1040 pts/3     S    09:41   0:00 bash
...
```

Another common use is to remove certain lines from a file. To remove the user nicki from the file /etc/passwd, you can do this:

```
# grep -v '^nicki' /etc/passwd > /etc/passwd.new
# mv /etc/passwd.new /etc/passwd
```

We should mention that this is not the most robust method of removing a user. If the grep command failed for some reason (maybe the drive is full), you should not copy the new file over the existing password file. A better way to run this command would be as follows:

```
# grep -v '^nicki:' /etc/passwd > /etc/passwd.new \
&& mv /etc/passwd.new /etc/passwd
```

Now, the file move will not occur unless the first command was successful. The main disadvantage of this method is that the permissions of the original file may be lost. You could fix the permissions after the modification (never a bad idea), or you can expand the command sequence to the following:

```
# grep -v '^nicki:' /etc/passwd > /etc/passwd.new \
&& cp /etc/passwd.new /etc/passwd \
&& rm -f /etc/passwd.new
```

Now, the new file is copied over the original, preserving the permissions of the original file. This still doesn't do any file locking, though. Somebody or something else could modify the password file during this process, and those changes would be lost. Usually, other cleanup is also necessary when you are removing a user.

Other command-line options are available. The `-i` switch makes the pattern matching case-insensitive. The `-l` switch lists the file names containing matching lines instead of printing the lines themselves. The `-r` switch available on some versions recursively follows directories.

The sed Stream Editor

`sed` is a stream editor, which means it can take an input stream and make modifications to that stream. As long as you understand the basics of regular expressions, a little bit of tinkering and reading of the man page should go a long way to help you understand `sed`. The power of the regular expression library is not as powerful as you have available to you in Perl (or even `egrep`), but it is sufficient to solve many problems.

Modifying a File

`sed` can operate on either standard input (`stdin`) or on files specified as arguments. The output of `sed` always comes out on the standard output (`stdout`). If you want to use `sed` to modify a file (a common task), you should first copy the file and then direct `stdout` to the original file. Once you are sure your `sed` command is correct, you can remove the copy. However, you can very easily create a `sed` command that will result in no output, so leave the copy there until you are absolutely sure nothing went wrong.

Here is an example of modifying a file with `sed`. We will first create a file containing the word `hello` and then use `sed` to remove all `l` characters:

```
$ echo "hello" > file.orig
$ sed 's/l//g' file.orig > file.new
$ cat file.new
heo
```

The `sed` command itself deserves some explanation. The entire pattern is enclosed in single quotes to avoid any problems with the shell modifying the pattern. The first character, `s`, is the command (substitute). The forward slash is used as a delimiter—it separates the various components of the substitute command. The first component contains the letter `l`, or the search string (or the regular expression in most cases). The next component contains the substitution string, which is empty in our case. Finally, the `g` at the end is a modifier for the substitute command that causes it to repeat the substitution as many times as necessary on each line because, by default, `sed` only performs the command once per line of input. So, the final result is that every occurrence of the `l` character in the original file has been removed by `sed` in the new file.

Modifying stdin

More often than not, sed is used to modify a stream on the standard input. Instead of specifying a file name, you simply pipe the text to be processed into sed using the shell pipe character (|). The previous example can be done in almost the same way using a pipe:

```
$ echo "hello" > file.orig
$ cat file.orig | sed 's/l//g' > file.new
$ cat file.new
heo
```

Or, in this case, we could bypass the file altogether. We echo the word "hello" directly into sed, and allow sed's output to go directly to the screen:

```
$ echo "hello" | sed 's/l//g'
heo
```

This is actually an excellent way to test sed commands. If a sed command within a shell script is giving you problems, you can always run it on the command line to see if the expression is working properly.

A more real-world use of sed would be to modify the first line of a Perl script to fix the path to the Perl interpreter. Let's say that your Perl interpreter is called as /usr/local/bin/perl. If a script is specified /usr/bin/perl, then you could use this sed command to replace that (or any other) path to the interpreter. It will also maintain any arguments to the interpreter. In the real world, you would run this command on a file, but here is the actual command with a few test cases that can be run directly on the command line:

```
$ echo '#!/usr/bin/perl' |
> sed 's=^#!.*perl=#!/usr/local/bin/perl='
#!/usr/local/bin/perl
$ echo '#!/opt/bin/perl -w' |
> sed 's=^#!.*perl=#!/usr/local/bin/perl='
#!/usr/local/bin/perl -w
```

As you can see, this command will change any path to the Perl interpreter to the correct one and also preserves arguments in the process. The period character (.) stands for any character, so .* will match zero or more of any character (i.e., any path before the string perl). Of more importance is the = character that immediately follows the s command—with sed, you can use any character as a delimiter. Since the replacement string contained several / characters (the standard delimiter), we chose another character to make things simpler.

Isolating Data

Within shell scripts, using `sed` to isolate certain portions of strings is common. If, for example, you want to determine the system's IP address from the output of the `ifconfig` command, you have to isolate the IP address from the following output:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:a5:5c:25:39:80
          inet addr:10.1.1.30  Bcast:10.1.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:33575 errors:0 dropped:0 overruns:0 frame:0
          TX packets:71702 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:17893725 (17.0 Mb)  TX bytes:11724172 (11.1 Mb)
          Interrupt:3 Base address:0x100
```

The first step is to isolate the proper line. You can use the `-n` command-line option to cause `sed` to not display any output, by default. You can then use the `p` option to print out only the lines that are matched:

```
$ ifconfig eth0 | sed -n '/inet addr:/p'
          inet addr:10.1.1.30  Bcast:10.1.255.255  Mask:255.255.0.0
```

You can then expand this command to also isolate only the data you desire:

```
$ ifconfig eth0 | sed -n 's/.*inet addr:\([^ ]*\).*/\1/p'
10.1.1.30
```

Now, you have isolated the system's IP address. If you were writing a shell script, you would want to store that value in an environment variable:

```
$ IP_ADDR=`ifconfig eth0 | sed -n 's/.*inet addr:\([^ ]*\).*/\1/p'`
$ echo $IP_ADDR
10.1.1.30
```

Other Tools

`sed` is not the only option for modifying streams of text. Other solutions are more powerful but generally more complicated. AWK can do everything `sed` can do and more. Perl can do everything AWK can do, and more. So, if you already know one of those languages, you can use them to do the same things you could do with `sed`.

sed Resources

You can find plenty of information on sed simply by reading the man page (by running man sed). You can also obtain a great reference for both sed and AWK by purchasing *sed and awk*, by Dale Dougherty and Arnold Robbins (O'Reilly Media Inc., 1997).

AWK

Although AWK is a full-fledged programming language used for text processing, we only use it for fairly simple tasks within this book. We prefer to use Perl for the more complicated work. For that reason, we provide only a brief overview here. For additional information, explore the resources suggested in the "AWK Resources" section.

Basic AWK is fairly standard across different operating systems. There is also the GNU version, gawk, which provides additional functionality. Both versions can commonly be found on most Linux systems.

Very Basic Usage

We often find ourselves using AWK as a glorified version of the cut command. The cut command can be used to isolate certain fields from each line of input. You can retrieve a list of usernames, for example:

```
$ cut -d: -f1 /etc/passwd
root
bin
daemon
...
```

Here, we simply requested a delimiter of : (-d:) and the first field (-f1). We can also do the exact same thing, using a different syntax, with AWK:

```
% awk -F: '{print $1}' /etc/passwd
root
bin
daemon
...
```

The -F: switch overrides the default delimiter to :. The {print $1} sequence is an actual AWK program, specified directly on the command line. It is executed on each line of input and simply prints out the first field of each line.

AWK is even more useful, though, when the fields are separated by arbitrary amounts of whitespace. The cut command can only look for a single delimiter, whereas the awk

command, by default, uses any sequence of whitespace as the delimiter (any number of spaces and tabs). Here is some example output from the command `ps auwx`:

```
$ ps auwx
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  1336  432 ?        S    09:39   0:04 init
root         2  0.0  0.0     0    0 ?        SW   09:39   0:00 [keventd]
root         3  0.0  0.0     0    0 ?        SW   09:39   0:00 [kapmd]
...
```

Let's say that we want a listing of all active process IDs:

```
$ ps auwx | awk '{print $2}'
PID
1
2
3
```

We have one problem, however. The `PID` string is part of the header line and should not be included in the output. We will address this issue in the next section.

Not-Quite-As-Basic Usage

Continuing from the example in the previous section, we will use a more complicated AWK command to eliminate the header from the process ID listing:

```
$ ps auwx | awk '!/^USER/ {print $2}'
1
2
3
```

The command is now preceded by a regular expression. The command only operates on lines that first satisfy the regular expression. In this case, the line must not begin with the string USER. This will be true of all lines except for the header line.

Now, we will use some contrived examples to illustrate some more functionality. It is standard practice on many systems to create a group for each user. Let's say that we wanted to know what system groups contained members other than the user who owns the group. Here are a few entries from `/etc/group`:

```
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
tty:x:5:
```

We want to ignore the root group because the user root is the only member. We want to ignore the tty group, because there are no specified members. The bin and daemon groups should be included in the output. Here is the program:

```
$ awk -F: '{if ($4 && ($1 != $4)) print $1}' /etc/group
bin
daemon
```

We can simplify the program by using a program file and the -f option:

```
$ awk -F: -f test.awk /etc/group
bin
daemon
```

where the file test.awk contains the program:

```
{
   if ($4 && ($1 != $4))
      print $1
}
```

All we are doing here is checking to see if field 4 contains something and that it is not equal to field 1. If both of these conditions are true, field number 1 is printed.

Much more power is available to you in AWK. You will see a bit of that power in the examples throughout this book. You can learn even more by reading the resources available outside of this book.

AWK Resources

Apart from the AWK man page, you can obtain a great reference by purchasing *The AWK Programming Language*, by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger (Addison-Wesley, 1988).

■ ■ ■

Writing cfengine Modules

Cfengine automatically sets a large number of classes at runtime based on attributes of the system. These are classes based on the IP address of the system, the operating system (e.g., `linux` or `solaris`), the date and time, and many other attributes. Many predefined cfengine classes are shown and explained in Chapter 4.

Cfengine modules are designed for the definition of custom classes. Modules allow you to write code to extend cfengine, so that it can detect new situations and site-specific conditions. We say "designed for" because it's possible to use modules to implement system changes as well. We'll focus on what modules are designed for and then briefly touch on other uses. We'll explain the requirements for using modules and then show you how to create a simple module to get you started. Once you know how to create and use a module, you'll be able to build on the example in your own environment.

Requirements for Using Modules

Before we discuss modules in any detail, we'll lay out the requirements for using them:

- Modules must be placed in the directory defined by the variable `moduledirectory` in `cfagent.conf` (or a file imported from `cfagent.conf`).

- Each module file must have a name following the convention `module:`*mymodule*.

- `cfagent` will execute only modules

  - Owned by root (or the user running cfagent)

  - Placed in the special directory

  - That follow the naming convention

- Modules may be written in any language supported on the system, and they can output anything you deem appropriate. The important things about module output follow:

  - Lines that begin with a + sign are interpreted as classes to be defined.

  - Lines that begin with a - sign are interpreted as classes to be undefined.

  - Lines starting with = are interpreted as variables to be defined.

- Any other lines of output are also printed by cfagent, so modules should generally be silent.

Defining Custom Classes Without Modules

Classes are used by cfengine to determine the appropriate actions to take, if any. During the development of our example environment used throughout this book, we only needed classes based on simple tests. For example, the following shellcommands section will only be run if the niagara_t1_proc class is set:

```
classes:
   sunos_sun4v::
      niagara_t1_proc =
         ( "/usr/platform/sun4v/sbin/prtdiag | grep UltraSPARC-T1 >/dev/null" )

shellcommands:
   niagara_t1_proc::
      "/bin/echo hello world"
```

Sun hardware classified as sun4v has the processor class that we're looking for but not *all* systems of that class run a particular CPU called the Niagara T1 processor. In the classes section, we ran the prtdiag command and piped the output into the grep command looking for the string "UltraSPARC-T1". Running the prtdiag and grep commands enabled us to find the sun4v systems that are running the Niagara T1 processor and then to set the niagara_t1_proc class.

This very simple example of setting a custom class is well suited to the classes section because of its simplicity. At some point, you may need to set classes based on much more complex criteria. If you can write code in any language supported on your systems, you can write a cfengine module to set your custom classes. We will use Bourne shell scripting for our example module.

Creating Your First cfengine Module

Making use of a module to set the `niagara_t1_proc` class is a good way to get familiar with creating your own modules. We will implement this simple module in our example environment.

First, we created a module called `module:detect_niagara`, with these contents:

```
1.  #!/bin/sh
2.  PATH=/bin:/usr/bin
3.
4.  if /usr/platform/sun4v/sbin/prtdiag | grep UltraSPARC-T1 >/dev/null
5.  then
6.          THREAD=`/usr/platform/sun4v/sbin/prtdiag | grep UltraSPARC-T1 \
7.          | wc -l | sed 's/^[ \t]*//' `
8.          echo "+niagara_t1_proc"
9.          echo "=num_cores=$THREAD"
10. fi
```

This script is very simple. It executes a `grep` command against the output of the `prtdiag` command on line 4, and if a match is found, three things happen:

1. On line 6, the `prtdiag` command is run again—this time to capture the total number of CPU threads present on the system's processor—using the `wc` command. The `sed` command in the pipeline removes any leading whitespace placed in the output by the `wc` command.

2. The `niagara_t1_proc` class is set using an echo statement on lines 7 and 8, so now, the `cfagent` process running this module will have the class defined.

3. A variable named `num_cores` will be passed back to the `cfagent` process running the module, with the value set to the number of threads on the system from line 6.

We placed the file in the `PROD/inputs/modules` directory, which should exist if you followed along with this book (this relative path convention has also been used throughout this book; the full path on the cfengine master in our example environment is `/var/lib/cfengine2/masterfiles/PROD/inputs/modules`). If not, you may need to create the directory. We added this line to `PROD/inputs/control/cf.control_cfagent_conf` so our module could be found by `cfagent` at runtime (make sure that it applies to the `any` class):

```
moduledirectory         = ( $(client_cfinput)/modules )
```

We then created a task at PROD/inputs/tasks/misc/cf.detect_niagara_proc with these contents:

```
control:
    sunos_sun4v::
        addinstallable    = ( niagara_t1_proc )
        actionsequence = ( module:detect_niagara )

shellcommands:
    sunos_sun4v.niagara_t1_proc::
        "/bin/echo hello world - I have a Niagara proc with $(num_cores) threads"
```

We needed the addinstallable line so that cfengine knew that a custom class might be defined. We set the actionsequence to include this module on any hosts running the sun4v architecture—recall that modules are always called via the cfengine actionsequence. We run the command in the shellcommands section when a host is a sun4v system and when the niagara_t1_proc class is set. When the command is run, the variable containing the number of threads on the processor is returned.

To put the task into use, we added it to the PROD/inputs/hostgroups/cf.any file with this entry:

```
tasks/misc/cf.detect_niagara_proc
```

Check the files into Subversion, and check them out onto your cfengine master, if applicable and if you've followed along with the book to this point (if you're not sure what we mean, don't worry about it). We don't need take any extra measures in our example environment to set the ownership or permissions on the files in our modules directory, because the update.conf in our example environment copies all files inside the PROD/inputs directory with root (user and group) ownership and other permissions completely absent (file permission mode 700 and owned by root:root). If you haven't followed along with this book, here's the pertinent section from update.conf:

```
copy:
                $(master_cfinput)/
                                    dest=$(workdir)/inputs/
                                    r=inf
                                    mode=700
                                    type=checksum
                                    ignore=RCS
                                    ignore=.svn
                                    owner=root
                                    group=root
```

```
ignore=*,v
purge=true
server=$(policyhost)
trustkey=true
encrypt=true
```

Our modules directory is under the inputs directory (which is copied via update.conf), and this copy action recursively copies all files and directories beneath the inputs directory. The variables used aren't pertinent to this section. What's important is that the module files are owned by root, since we only run cfengine as root at our example site.

On systems running a Niagara processor (such as a Sun T2000 system), you'll see output from cfagent like this:

```
cfengine:sunbox:/bin/echo hello: hello world - I have a Niagara proc with 32 threads
```

This simple example puts all the pieces in place for you to successfully use cfengine modules. You can use it to build a much more complicated module that sets classes and variables that you can then use to take actions in a cfengine task file (as we did with the echo command in the shellcommands section of cf.detect_niagara_proc).

Using modules, you can extend cfengine in ways never imagined by the author of cfengine.

Using Modules in Place of shellcommands

Cfengine provides the shellcommands section so you have an easy way to perform custom actions. The commands defined in a shellcommands section can be standard operating system utilities or custom scripts. Cfengine makes every attempt to be as generic as possible, and it directly supports only the most basic system administration actions (e.g., file copies, permission fixes, link creation, file editing, etc).

Nothing prevents the code in a module from making changes on a system. The entire list of classes defined by cfengine on the host is passed to scripts or programs run by shellcommands as well as to scripts or programs run as a module, so there is no technical barrier to using a module instead of a shellcommands section.

We don't like to use modules this way, because they weren't designed to replace shellcommands. We think that some sites choose to use modules in place of shellcommands since it's easy to automate the copy of the modules directory and use that as a single location for cfengine-specific scripts. In our example environment, we automated the copy of an administrative script directory, so we have an easy location to place sitewide scripts for execution by administrators, cfengine, or both.

Modules are sometimes recommended on Internet mailing lists when the quotes used in `shellcommands` actions get too complicated for the cfengine parser, resulting in errors. Consider a `shellcommands` section such as this:

```
shellcommands:
    debian.i686::
        "/bin/grep 'foo bar' /etc/foobar | /bin/sed 's/^[ \t]*//'  \
            | /usr/bin/mail -s\"file contents from `hostname`\" recpient@example.org"
```

This code is difficult to read and needs some escaping of double quotes, and sometimes, cfengine can get confused when parsing commands like this. However, we don't consider this a reason to put the commands in a cfengine module. Instead, use a shell script, in a location such as our example site's `/opt/admin-scripts` directory.

We could create `PROD/repl/admin-scripts/mail-foobar` with these contents:

```
#!/bin/sh
PATH=/bin:/usr/bin

if [ -f /etc/foobar ]
then
    /bin/grep 'foo bar' /etc/foobar | /bin/sed 's/^[ \t]*//'  | \
      /usr/bin/mail -s"file contents from `hostname`" recpient@example.org
fi
```

We could then create a new `shellcommands` section like this:

```
shellcommands:
        debian.i686::
                "/opt/admin-scripts/mail-foobar"
```

Now, the cfengine configuration is easy to read, and the script can be a simple shell script with no special escaping rules, making it easy to read too. You can also easily add extra niceties to the shell script, like a `PATH` statement and a test for the `/etc/foobar` file's existence before running `grep` against it. At our example site, the contents of `PROD/repl/admin-scripts` on our cfengine master are copied to `/opt/admin-scripts` on all hosts, so once we place a script into the central directory, the copy is automatic.

We definitely recommend using shell scripts—not modules—for complicated `shellcommands` sections.

Index