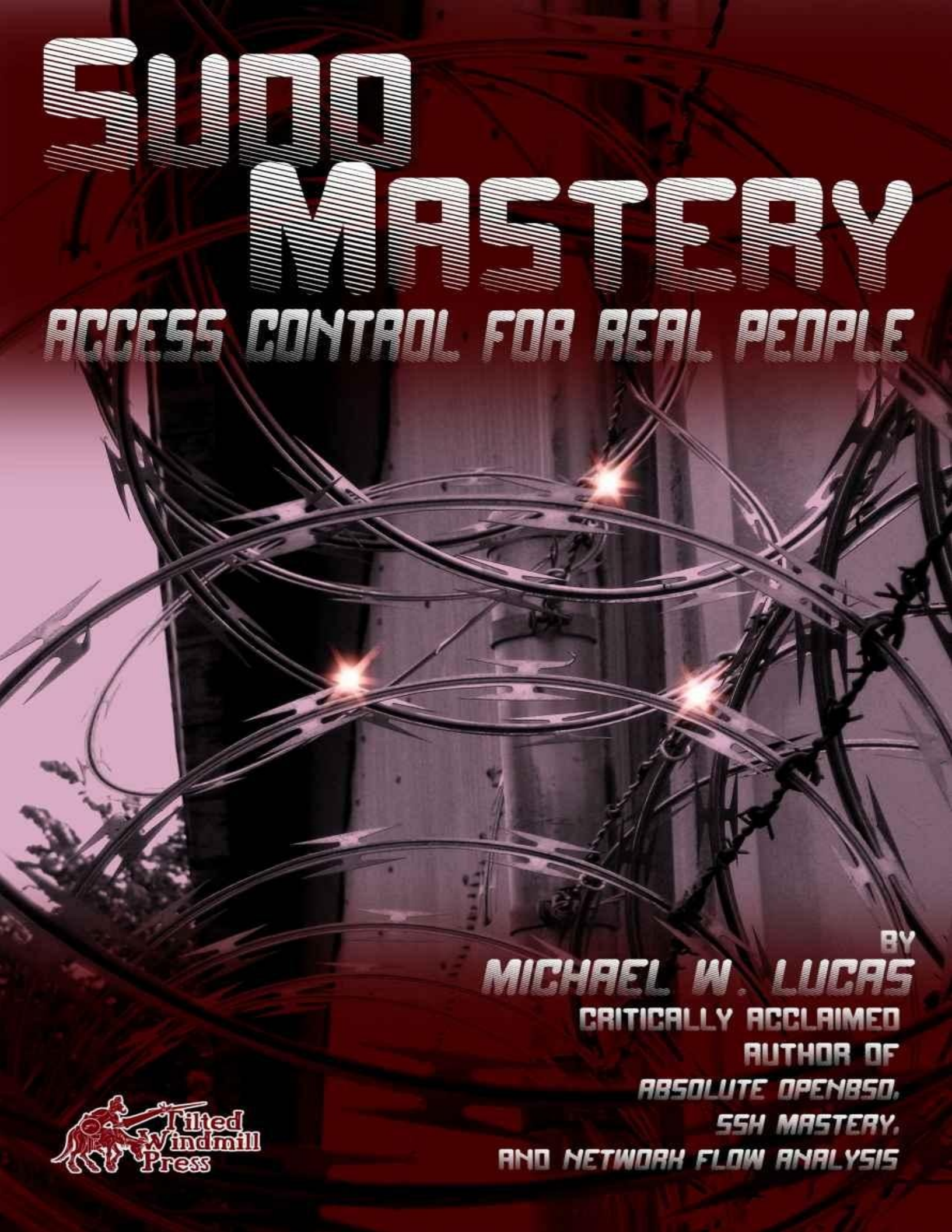
SUDO
MASTERY
ACCESS CONTROL FOR REAL PEOPLE
BY
MICHAEL W. LUCAS
CRITICALLY ACCLAIMED
AUTHOR OF
ABSOLUTE OPENBSD,
SSH MASTERY,
AND NETWORK FLOW ANALYSIS
Tilted
Windmill
Press

Sudo Mastery:

User Access Control for Real People

by Michael W Lucas

Tilted Windmill Press

Praise for other books by Michael W Lucas

Absolute OpenBSD, 2nd Edition

"Michael Lucas has done it again." – *cryptednets.org*

"After 13 years of using OpenBSD, I learned something new and useful!" – *Peter Hessler, OpenBSD Journal*

"This is truly an excellent book. It's full of essential material on OpenBSD presented with a sense of humor and an obvious deep knowledge of how this OS works. If you're coming to this book from a Unix background of any kind, you're going to find what you need to quickly become fluent in OpenBSD – both how it works and how to manage it with expertise. I doubt that a better book on OpenBSD could be written." — *Sandra Henry-Stocker, ITWorld.com*

"Do you need this book? If you use OpenBSD, and have not yet achieved guru status, yes, this book is just for you. Even gurus will find valuable things in this book that they did not know… But

beyond the OpenBSD aspect, there are great sections on cross-platform applications like *sudo* that are almost enough on their own to justify getting this book. And there are several of those chapters. So: even if you don't use OpenBSD directly, would you like a quick reference on *sudo*, IPv6 networking, and NFS setup? Oh, and also *tftpd*, PXE, and diskless BSD systems? But wait, what if I told you these references came with a free book on OpenBSD installation and configuration?" – *Warren Block, wonkity.com*
"It quickly becomes clear that Michael actually uses OpenBSD and is not a hired gun with a set word count to satisfy... In short, this is not a drive-by book and you will not find any hand waving." – *Michael Dexter, callfortesting.org*

DNSSEC Mastery
"When Michael descends on a topic and produces a book, you can expect the result to contain loads of useful information, presented along with humor and real-life anecdotes so you will want to

explore the topic in depth on your own systems." — *Peter Hansteen, author of The Book of PF*

"Pick up this book if you want an easy way to dive into DNSSEC." — *psybermonkey.net*

SSH Mastery

"…one of those technical books that you wouldn't keep on your bookshelf. It's one of the books that will have its bindings bent, and many pages bookmarked sitting near the keyboard." — *The Exception Catcher*

"…SSH Mastery is a title that Unix users and system administrators like myself will want to keep within reach…" — *Peter Hansteen, author of The Book of PF*

"This stripping-down of the usual tech-book explanations gives it the immediacy of extended documentation on the Internet. Not the multipage how-to articles used as vehicles for advertising, but an in-depth presentation from someone who used OpenSSH to do a

number of things, and paid attention while doing it." — *DragonFlyBSD Digest*

Network Flow Analysis

"Combining a great writing style with lots of technical info, this book provides a learning experience that's both fun and interesting. Not too many technical books can claim that." — *;login: Magazine, October 2010*

"This book is worth its weight in gold, especially if you have to deal with a shoddy ISP who always blames things on your network." — *Utahcon.com*

"The book is a comparatively quick read and will come in handy when troubleshooting and analyzing network problems." —*Dr. Dobbs*

"Network Flow Analysis is a pick for any library strong in network administration and data management. It's the first to show system administrators how to assess, analyze and debut a network using

flow analysis, and comes from one of the best technical writers in the networking and security environments." — *Midwest Book Review*

Absolute FreeBSD, 2nd Edition

"I am happy to say that Michael Lucas is probably the best system administration author I've read. I am amazed that he can communicate top-notch content with a sense of humor, while not offending the reader or sounding stupid. When was the last time you could physically feel yourself getting smarter while reading a book? If you are a beginning to average FreeBSD user, Absolute FreeBSD 2nd Ed (AF2E) will deliver that sensation in spades. Even more advanced users will find plenty to enjoy." — *Richard Bejtlich, CSO, MANDIANT, and TaoSecurity blogger*

"Master practitioner Lucas organizes features and functions to make sense in the development environment, and so provides aid and comfort to new users, novices, and those with significant

experience alike.” — *SciTech Book News*

“…reads well as the author has a very conversational tone, while giving you more than enough information on the topic at hand. He drops in jokes and honest truths, as if you were talking to him in a bar.” — *Technology and Me Blog*

Cisco Routers for the Desperate, 2nd Edition

“If only Cisco Routers for the Desperate had been on my bookshelf a few years ago! It would have definitely saved me many hours of searching for configuration help on my Cisco routers. . . . I would strongly recommend this book for both IT Professionals looking to get started with Cisco routers, as well as anyone who has to deal with a Cisco router from time to time but doesn’t have the time or technological know-how to tackle a more in-depth book on the subject.” — *Blogcritics Magazine*

"For me, reading this book was like having one of the guys in my company who lives and breathes Cisco sitting down with me for a

day and explaining everything I need to know to handle problems or issues likely to come my way. There may be many additional things I could potentially learn about my Cisco switches, but likely few I'm likely to encounter in my environment." — *IT World*
"This really ought to be the book inside every Cisco Router box for the very slim chance things go goofy and help is needed 'right now.'" — *MacCompanion*

Absolute OpenBSD

"My current favorite is Absolute OpenBSD: Unix for the Practical Paranoid by Michael W. Lucas from No Starch Press. Anyone should be able to read this book, download OpenBSD, and get it running as quickly as possible." — *Infoworld*
"I recommend Absolute OpenBSD to all programmers and administrators working with the OpenBSD operating system (OS), or considering it." — *UnixReview*
“Absolute OpenBSD by Michael Lucas is a broad and mostly

gentle introduction into the world of the OpenBSD operating system. It is sufficiently complete and deep to give someone new to OpenBSD a solid footing for doing real work and the mental tools for further exploration… The potentially boring topic of systems administration is made very readable and even fun by the light tone that Lucas uses.” — *Chris Palmer, President, San Francisco OpenBSD Users Group*

PGP & GPG

"...The World's first user-friendly book on email privacy...unless you're a cryptographer, or never use email, you should read this book." — *Len Sassaman, CodeCon Founder*

“An excellent book that shows the end-user in an easy to read and often entertaining style just about everything they need to know to effectively and properly use PGP and OpenPGP.” — *Slashdot*

“PGP & GPG is another excellent book by Michael Lucas. I thoroughly enjoyed his other books due to their content and style.

PGP & GPG continues in this fine tradition. If you are trying to learn how to use PGP or GPG, or at least want to ensure you are using them properly, read PGP & GPG." — *TaoSecurity*

Sudo Mastery

Sudo Mastery: User Access Control for Real People

Amazon Edition.

Author: Michael W Lucas
Cover design: Bradley K McDevitt
Copyediting: Aidan Julianna "AJ" Powell
Cover Photo: Elizabeth Lucas (concertina wire at abandoned factory, Detroit)

published 2013 by Tilted Windmill Press
www.tiltedwindmillpress.com

For Liz

Acknowledgements

I want to thank the folks who reviewed the manuscript for Sudo Mastery before publication: Bryan Irvine, JR Aquino, Hugh Brown, and Avigdor Finkelstein. Special thanks are due to Todd Miller, the current primary developer of sudo, who was very patient and helpful when answering my daft questions.

While I appreciate my technical reviewers, no errors in this book are their fault. All errors are my responsibility. Mine, do you hear me? You reviewers want blame for errors? Go make your own.

XKCD fans should note that the author does not particularly enjoy sandwiches. However, Miod Vallat, currently exiled to France, would really like a sandwich with nice fresh bread, really good mustard, and low-carb ground glass and rusty nails. And

Bryan Irvine would like a rueben.

This book was written while listening obsessively to Assemblage 23.

Contents

Chapter 1: Introducing sudo

Resolved: controlling user access to a computer's privileged programs and files is a right pain. None of the systems that evolved to cope with mapping real-world privileges onto digital schemes are very good. The best access control systems merely hurt less than others.

Unix-like systems control programs and file access through users and groups. Each individual user has a unique identifier, given either as a username or a user ID number (UID). Users are arranged in uniquely identified groups, given either as a group name or a group ID number (GID). Specific users and groups have permission to access specific files and programs.

This scheme sufficed during UNIX's childhood. A large university might have a couple of UNIX servers. Hundreds of users logged onto each server for mail, news, and computation-intensive applications. Students went in one group, grad students in another,

then professors, staff, and so on. Individual classes and departments might have their own groups.

The system owners had a special account, **root**. The **root** account has ultimate system control. As a security and stability precaution, Unix-like systems restrict certain operations so that only **root** can perform them. Only **root** can reconfigure the network, mount new filesystems, and restart programs that attach to privileged network ports. This made sense when you had two servers for an entire campus – reconfiguring the network or adding a new disk drive is a serious task in that environment. The job of managing multimillion-dollar systems should remain in trusted, highly skilled hands.

In the 21st century, Unix-like systems are cheap and plentiful. Teams of people might share systems administration tasks, or one person might have complete control over a system, or anything in between. Either situation completely changes your security requirements from those of the previous century.

Large organizations often divide systems administration responsibilities between skilled individuals. One person might be responsible for care and feeding of the operating system, while a second person handles the application running on the server. The server supports the application, and the application is why the server exists, but both people need to perform tasks that require root-level privileges. But root-level privilege is an all-or-nothing affair. There's no division between "access to change the kernel" and "access to run privileged applications." If the application administrator has root-level access, he can change the kernel. You can always rely on gentleman's agreements to only touch the parts of the system you're responsible for, but when your organization employs a team of systems administrators and a team of database administrators to support dozens or hundreds of servers, these gentleman's agreements quickly decompose into finger-pointing bloodbaths – even without vendor-provided application setup scripts that helpfully customize the kernel without telling anyone.

These organizations need a finer-grained access control system than **root** provides.

The all-or-nothing model breaks down even more when everyone has a Unix-like system. Setting aside the innumerable phones and tablets which have extra software to make them user-friendly, many folks run Unix-like operating systems on a desktop or laptop. Every time they access a USB drive or use a coffee shop wireless network, something on the system needs root-level privileges. Using root privileges isn't terribly onerous – log in with your regular account, use the `su` command to switch users, enter the root password, run the commands that need root access, and exit the root account. But when you must use the root account any time you put in a USB drive, bounce the network, add, reconfigure, or restart software, it quickly becomes downright annoying. While software can manage much of this for you, sometimes you must trigger root privileges for routine tasks.

The computing industry is full of really smart people that have

expanded the classic UNIX privilege control models. One method is through *setuid* and *setgid* programs. While programs normally work with the privileges of the user who runs them, setuid and setgid programs change their effective UID and GID to some other value. You can have a setuid program that runs as root. Changing your password requires editing secured files in */etc/*, so the passwd command is setuid. But intruders really like setuid and setgid programs. Flaws in these programs might be exploited into full root access. And most operating systems don't let you make shell scripts setuid, only programs.

Then there are several varieties of *access control lists* (ACLs) which more broadly expand the user-group-others ownership model. ACLs allow you to declare something like "This person owns the file, but *these* groups and people can modify it, with *these* exclusions, and *these* groups and people (with some exclusions, of course!) can execute it, while these other people can read data from it, except for…" At this point the systems administrator gets a

headache and starts contemplating a career cleaning up real sewage instead of the metaphorical kind. And of course, all the different ACL implementations are ever so slightly incompatible. Very few people can correctly implement ACLs on a single platform, and that expertise doesn't really extend to other platforms. ACLs have a place in systems administration, and if you really need them, they're invaluable. But most of us don't need them.

And sadly, access control lists are about as good as it gets.

Except for sudo.

What Is Sudo?

Sudo is a program that controls access to running commands as root or other users. The system owner creates a list of privileged commands that each user can perform. When the user needs to run a command that requires root-level privilege, he asks `sudo` to run the command for him. Sudo consults its permissions list. If the user has permission to run that command, it runs the command. If the user does not have permission to run the command, sudo tells him so. Running `sudo` does not require the root password, but rather the user's own password (or some other authentication).

The system administrator can delegate root-level privileges to specific people for very specific tasks without giving out the root password. She can tell sudo to require authentication for some users or commands and not for others. She can permit users access on some machines and not others, all with a single shared configuration file.

Some applications, notably big enterprise database software, run

under a specific dedicated account. Users must switch to this account before managing the software. You can configure sudo to permit users to run specific commands as this account. Maybe your junior database administrators only need to run backups, while the lead DBA needs a full-on shell prompt as the database account. Sudo lets you do that.

Finally, sudo logs everything everybody asks it to do. It can even replay the contents of individual `sudo` sessions, to show you exactly who broke what.

What's Wrong with Sudo?

If sudo is so awesome, why doesn't everybody use it?

Sudo adds another layer of systems administration. Adding that layer requires time, energy, and attention. It requires learning yet another danged program when you already have too much to do. If you're responsible for running an enterprise system with several groups of administrators, investing in sudo reduces your workload. But you must learn how to use it first.

Some commercial UNIXes don't include sudo because they already include their own proprietary escalated privilege management system. OpenSolaris-based systems have `pfexec` and role-based access control (RBAC). HP has `pbrun`. If you were a commercial UNIX vendor who spent lots of money and energy developing an ACL-based privilege management system, would you include and encourage use of a simpler, easier tool instead? I might, but that's why I'm not a big commercial UNIX vendor.

Many open-source Unix-like operating systems do include sudo

in their base system. Some, such as Ubuntu and OS X, completely disable the **root** account and only permit privileged access via sudo. This is a lurch in the right direction, but most people who have sudo use it incorrectly.

What's the wrong way to use sudo? Sudo is not a replacement for su. Sudo is not a way to completely avoid requiring authentication for privileged access. Sudo is not a tool to force someone to make you a sandwich. A proper sudo setup simplifies system management. An improper sudo setup lets intruders and unauthorized users corrupt or destroy your system faster and easier.

"Proper use of sudo" doesn't mean complicated, or even extensive policies. I've seen system administrators spend hours writing complicated sudo policies, only to watch users waltz right past their restrictions. Sometimes the users didn't even realize that the restrictions were in place. Sudo has limits. Once you understand those limits, you can make realistic decisions about

how and where your organization deploys sudo.

The problem I see most often with sudo has nothing to do with the software itself. A proper sudo deployment in a complicated organization requires the system administration team to agree who is responsible for what. Sudo enforces job duties and responsibilities in a configuration file. The configuration file is flexible, but people cannot exceed the privileges specified therein.

What are the boundaries of your responsibilities? What permissions do you need to do your real job, and which tasks should someone else do? Being forced to sit down and think about these things can be uncomfortable, and can temporarily increase conflicts within an organization. Once the arguments settle, however, conflicts decrease. There's no bickering over who did what, when, or how. Everybody knows that the database team can't format filesystems, the web team can't restart the database, and the sudo logs clearly show who took any privileged actions. And having an audit trail improves system stability. When people know

that the system logs their privileged actions, and that they can and will be held responsible for breaking things, they stop breaking things so often. Weird.

Who Does Sudo Protect You From?

Sudo protects the system from harm by intruders or systems administrators, and it protects systems administrators from many management problems.

Giving a user access to only a limited set of privileged commands limits the damage that user can inflict on the system. The user who only has access to manage the web server or database cannot mangle disk partitions. If an intruder compromises that user's account, the intruder is likewise slowed or contained.

Similarly, lack of access protects the system administrator when something goes wrong. Even without sudo logs, a user with limited administrative access can say "Hey, I didn't reconfigure the web server. I don't have that access, remember?" Accountability works both ways. Use it to your advantage.

Sudo Support

Sudo is freely-available open source software. You are welcome to download it from the main web site (http://sudo.ws) or a mirror and use it throughout your organization at no charge. The license permits you to use sudo as the basis of your own products, resell it to clients, or incorporate it into software you then redistribute or resell. You can use sudo for anything you like.

What you don't get is sophisticated support.

Sudo is not created by a commercial company. It's developed and supported by the users who need it, and coordinated for the last several years by Todd Miller. You can contribute to sudo by submitting patches and bug reports. You can find people and companies who will support your sudo install, and who will even write custom code for you. But there's nobody for you to yell at if your sudo install doesn't work the way you expect. There's no toll-free number, no minimum-wage support minion with a questionable grasp of your language waiting to take abuse and

invective in exchange for cash.

Having said that, the people on the sudo mailing lists are both extremely helpful and very interested in real problem reports. They respond well to requests for help and poorly to demands. If you want to demand help – if you want to scream and rant and rave and turn blue in the face until your problem goes away – any number of companies will sell you that.

The software is free. Sudo's "official support" is a gift that evaporates as soon as you stop treating it like one.

Who Should Read This Book?

Everyone who works on a Unix-like system should understand sudo.

If you are a system administrator responsible for maintaining a complicated system, you probably want to assign your application administrators exactly the privileges needed to do their jobs, no more and no less. Correct sudo configuration frees up your time and protects the system from well-intentioned disasters.

If you are an application administrator, you need to do your job. This means you need the access to perform privileged tasks. Working via sudo means changing your processes slightly – not in any major way, but you can go completely bonkers trying to figure out why `sudo cd` doesn't work until you understand what's really happening. An understanding of sudo lets you draft the sudo rules you need and give them to the system administrator. Even if the system administrator disagrees, negotiating in sudo policy language means that you both understand exactly what you're

requesting. You can have specific discussions about who is responsible for what. No system administrator will tell an application administrator that he doesn't need the access to manage his application – she certainly doesn't want that job! The only question is: how can that access be best accomplished?

If a disagreement between teams is broad enough, this is where you invoke management to make a very specific decision and set clear lines of authority and responsibility. In some environments, getting that manager to take that step is a miracle in itself. But a mandate to implement sudo lets you corner him. And if you have a cranky system administrator who claims that granting you necessary access without giving you root is impossible, this book will let you categorically refute that. Which, admittedly, is its own vindictive pleasure.

If you maintain only your personal system, why would you care about sudo? Even on a personal laptop, some commands merit more thought and consideration than others. I can understand

wanting to trivially reconfigure the laptop's network, tweak removable media, or kill that berserk web browser. You probably do these tasks so often that you understand them well – my fingers can configure a network card without disturbing my brain. But tasks you perform less often, such as installing software or formatting disks, require a little more attention. It makes sense to permit sudo to run routine tasks without a password, but to require authentication before upgrading.

Server Prerequisites

This book assumes you're running sudo on a Unix-like operating system. Sudo is available for BSD and Solaris derivatives, Linux, and every commercial UNIX. While my reference platform is FreeBSD, sudo works on all of these systems and more.

My reference implementation is sudo 1.8.8. If you're running an older version, some features might be absent. A surprising number of operating system vendors include wildly obsolete sudo packages. Check the version of sudo on your system by running `sudo –V`. If your version is much older than 1.8.8, upgrade. You can always get the latest source code and a selection of precompiled packages at the main sudo web site, http:// sudo.ws.

The sudo documentation and this book assume that your operating system conforms fairly closely to the traditional filesystem layout. The examples in this book show commands in standard directories such as */bin*, */usr/bin*, */sbin*, and so on. If your operating system uses its own directory layout, you'll need to

adjust the examples to match.

Sysadmin Background

Where many important programs require an extensive background in related software before you can use them, sudo is nice in that it's fairly self-contained. You can master sudo without understanding all the programs that users can access through sudo. Sudo is a system management tool, however; the more you understand your system, the better you can leverage sudo and the more confidence you'll have in your configuration. I assume you can install sudo, either from an operating system package or from source code.

Configuring sudo requires root access on a Unix-like system and familiarity with a terminal-mode text editor. Sudo defaults to using vi, but you can use Emacs or any other editor.

That's everything. Really. All the other knowledge you'll pick up as we go.

Learning Sudo

The goal of this book is to let you replace access to privileged commands via the `su` command and the root password with the `sudo` command and the user's personal authentication credentials. Once you're comfortable with sudo, you can use the system's authentication mechanism to eliminate a user's ability to become root via `su`. The root password will become something only used in a disaster, or perhaps when you're at the physical console. Eliminating root's generic authentication improves systems administrator accountability within an organization. But one of the best ways to make this project fail is to deploy sudo too quickly.

Configuring sudo has its own pitfalls. You'll need to learn how sudo fits into your environment. Nothing causes quite as much agonized, frustrated self-recrimination as locking yourself out of your own server. Don't be too quick to disable root access via su, as you can use that access to repair a broken sudo configuration. Yes, sudo has features and tools to verify that your sudo policy is

syntactically correct. A sudo policy that says "nobody can do anything" is syntactically correct, however. Leave your old root access in place until you're absolutely confident in the new sudo arrangements, or become comfortable with booting your system into single-user mode to repair either `su` or `sudo`. A virtual machine or jail can be a useful tool for destructive learning.

Some operating systems (notably Ubuntu and OS X) provide root access with `sudo` rather than `su`. If you're experimenting with sudo, and sudo is your main method of accessing privileged commands, you're in a risky situation. Before mucking with sudo, enable root access and put a root password on your learning machine. Make sure it works and that you can get root access without sudo. You can then freely explore sudo without blocking privileged access. Once you're comfortable with sudo, you can fully deploy it without worry.

The official sudo documentation describes various sudo features in Extended Backus-Naur Form (EBNF), a formal grammar for

program configuration. While familiarity with EBNF is a useful skill for any sysadmin, I'm choosing to not take you through the formal definitions. Instead, this book demonstrates the most important sudo features through snippets of actual configuration policies.

Also note that this book does not cover all possible sudo configurations, nor does it cover every available sudo feature. I cover what the vast majority of systems administrators need, but if you're running an older operating system, using an old version of sudo, or administer a Unix-like system that veers wildly from the common standards, you'll need to dive into the documentation to identify the sharp edges of your situation. But after reading this book you'll have a solid grounding in sudo techniques and a good idea of exactly what information you're looking for and how to use it.

Avoiding sudo

Many system administrators configure their systems to require root privileges for routine tasks when they should use the group privileges supported by the base operating system. We tend to look at permissions for the user and others, but pay less attention to group permissions. Before running to sudo to solve an access problem, see if you can solve your problem with groups instead. Requiring root privileges to permit access to files or programs is like requiring use of a sledgehammer to hang a picture.

Use group permissions for programs or files that need to be accessed by several people, and only those people. As a trivial example, assume several people maintain the files for a Web site. You can create a group called, say, **webadmins**, and assign that group as the owner of the web site directory and all files in it. Take a look at our web site's top-level directory.

```
# ls -l
total 94
drwxrwxr-x 2 mike  webadmins    512 Jul 12  2013 content
```

```
-rw-rw-r--  1 thea webadmins  16584 Oct 20  2013 logo.jpg
-rw-rw-r--  1 pete webadmins    767 Oct 20  2013 errata.html
-rw-rw-r--  1 mike webadmins   2736 Jul 12  2013 index.html
-rw-rw-r--  1 pete webadmins    167 Jul 12  2011 index2.html
-rw-rw-r--  1 thea webadmins  66959 Oct 20  2006 banner.jpg
```

The individual files are owned by a single person – **mike**, **thea**, or **pete**. But the files are also readable and writable by the group **webadmins**. Anyone in this group can read and edit these files, and anything in the directory beneath this one.

The specifics of adding groups varies among operating systems. I would tell you to edit */etc/groups*, but some operating systems have special tools specifically for overcomplicating – er, managing – groups. Use the tool recommended in your operating system manual.

What Groups am I In?

To identify which groups you are a member of, run id(1).

```
# id
uid=1001(mike) gid=1001(mike) groups=1001(mike),10020(webadmins)
```

My user account is in the groups **mike** and **webadmins**. I could edit the files in the example above based on my group membership. I could also edit a couple of those files because I own them and the permissions let the owner edit the files.

Programs versus Groups

Group permissions won't solve all access problems for programs. Some programs perform privileged functions, and letting a group run the program won't give the program the rights to perform the task. Remember, a program runs with the privileges of the person running the program.

To continue the saga of our web management team, web servers run on TCP ports 80 and/or 443. Only **root** can attach to network ports below 1024. If a user runs the web server program without any extra privileges, the program will run as that user account. It won't have the necessary privileges to attach to those network ports, and so the web server cannot start properly. Setting the

program permissions so that a user can run the program doesn't mean that the program will work. If you want your **webadmins** group to get root privileges specifically for starting, stopping, and otherwise managing the web server software, you need to give the users in that group root privileges. That's where sudo comes in – you can assign the members of the web management team control of the web server without giving them anything else.

Book Overview

Sudo is a suite of interrelated programs. You'll get better results configuring sudo if you understand how these different parts fit together.

Traditional sudo has two components: the sudo program and the sudoers policy engine. Chapter 2 gives you an elementary grounding in both.

The sudoers policy file can only be edited with root privileges. An error in the sudoers file prevents anyone from getting root privileges with sudo. If you've disabled root access through other means, a sudoers error locks you out of the system. The sudo suite includes a special tool, visudo, just for editing and validating the sudoers file. Using visudo reduces the odds you'll get really angry with yourself. I cover visudo in Chapter 3.

Sudo policies quickly become very complicated. Reduce this complexity through using lists and aliases, as discussed in Chapter 4.

You cannot adjust all parts of sudo's behavior through policy rules, however. The sudoers policy engine also includes various default settings and options to change them, which I detail in Chapter 5.

Some programs offer ways to break out of sudo's restrictions through shell escapes – not because they were written deliberately to avoid sudo, but because of their very nature. Chapter 6 covers ways to prevent getting an unrestricted root shell from text editors and similar programs.

Most of this book is about the sudoers security policy engine, but the sudo program itself can also be tweaked. Chapter 7 discusses *sudo.conf*.

A user's environment can cause all kinds of trouble when used by privileged programs. Chapter 8 covers cleaning the shell environment and either blocking or permitting environment variables in a sudo context.

Sudo can perform basic integrity checking on programs before

running them. You'll see how in Chapter 9.

Rather than maintaining a separate security policy on each of dozens or hundreds of machines, you can use one central policy and push it out to all your hosts. Chapter 10 covers realisticly using a single policy across your network.

Sudo can also get its security policy from your LDAP authentication server, rather than through the sudoers file. I cover LDAP and sudo in Chapter 11.

Once you control a user's access to privileged commands, the next question becomes "what did the user do?" Sudo includes three different logging systems, each with a different use case. Chapter 12 discusses all three.

Finally, sudo can treat user credentials in a variety of ways, and if you want to adjust how your sudo install handles passwords and other authentication data, you will want to read chapter 13.

But before we get to that advanced stuff, let's start with the basics about sudo.

Chapter 2: sudo and sudoers

The two key components of the sudo suite are the sudo program and */etc/sudoers*. Use the `sudo` command to run a program with escalated privilege. The sudoers file defines the policy telling sudo which commands a user can run, and with which privileges.

sudo 101

You want to run a command under sudo? Run `sudo` followed by the command. Here I ask for an NFS mount.

```
$ sudo mount fileserver:/home/mike /mnt
Password:
mount_nfs: fileserver: hostname nor servname provided, or not known
$
```

Sudo asks for a password. This is my password, not the root password. If I enter my password correctly, and if I have permission to run this command via sudo, I'll get the program's normal output. And about now is when I remember that the office support team renamed that machine.

The good news is, sudo remembers that I authenticated, and won't ask for my password for the next five minutes. Some operating systems change this time, so you'll want to check the sudo man page for details. (You can change this time, or use entirely different authentication, as discussed in Chapter 13.) If you make a mistake, you can reenter a corrected command

immediately afterwards and not have to retype the password.

The first time you run sudo on any system, sudo prints a few lines about the importance of thinking before you run privileged commands. Take this lecture to heart. Privileged commands are privileged because they can reconfigure, deconfigure, damage, demolish, or destroy a system.

Running Commands as Another User

Running commands as root isn't always desirable. Some software, notably databases and application servers, might have a dedicated user just for themselves. The application expects to run as that user, and that user's environment is configured to manage the application. Applications ranging from big Java programs to tiny tools such as Ansible[1] use this model. You can run a command as a specific user by adding the –u flag.

```
# sudo –u oracle sqlplus
```

This starts up the target user's environment and runs the

specified command, much like su -.

Running Commands as Another Group

Every user has a primary group, listed with their account in */etc/passwd* or its equivalent. Groups from additional sources, such as */etc/group*, are considered secondary groups. Some programs only work if the user's primary group is its preferred group. This gets really, *really* annoying, as you would probably prefer to use groups for their intended purpose rather than babysitting one piece of picky software. Depending on how your operating system handles groups and how your software is installed, you might need to change your primary group to run a command. Use the –g flag and a group name.

sudo –g operator stupidpickycommand

Sudo lies to the program and tells it that your primary group is **operator.**

You could also use a group ID number, by putting a hash mark before the GID. Your shell might demand you escape the hash

mark on the command line. We tcsh users don't have that requirement.

sudo –g #100 stupidpickycommand

Sudo runs the command as if your primary group ID is 100.

This is as much as 90% of users know about sudo. Everything else you learn will make you more of an expert.

sudoers 101

If running sudo seems simple, it's because the real work takes place in the sudoers file, often called just "sudoers." The sudoers file contains the rules defining which users can run which privileged commands. My examples assume that sudoers is */etc/sudoers*, but wherever your package puts it is fine. Never edit the sudoers file by hand; always use visudo as covered in Chapter 3.

Some operating system packages include OS-specific examples in their sudoers file for special features that the operating system supports. Before making any changes to the default sudoers file, copy the original to a safe location so you can refer to it later.

The sudoers file contains a series of rules, one rule per line. Every rule uses this general format. Most of the rest of our discussion on sudoers covers extending, stretching, and generally abusing this format.

username host = command

Username is the username that this rule applies to. The

username might also be a system group, or an alias defined within sudoers.

Host is the hostname of the system this rule applies to. We will share */etc/sudoers* across multiple systems in Chapter 10.

The equal sign separates the machine from commands.

Finally, *command* lists the full path to each command this rule applies to. Sudo configuration requires full paths to commands.

The sudoers file recognizes a variety of special keywords. One of the most commonly seen is ALL, which matches every possible option. To allow all users to run any command on every host, you could write a sudoers file like this:

```
ALL ALL=ALL
```

This is roughly equivalent to giving everyone root access, but using their own password instead of the root password. Don't do this. At a minimum, restrict access by username.

```
mike ALL=ALL
```

The user **mike** can run any command on all servers.

You can also restrict sudo access by host. Most commonly

you'll see the server limitation as ALL because most systems administrators configure sudo on a per-host basis. If you separately manage every server, defining the server as ALL really means "this server." As a best practice, however, put the server name here. (Run hostname to get the server's name.) Chapter 10 covers in detail assigning sudo privileges by host.

```
mike www=ALL
```

The user **mike** can run any command on the host **www**.

To restrict a user to running a single command, give the full path to the command in the last field.

```
mike www=/sbin/reboot
```

The user **mike** can run the command /sbin/reboot on the server **www**.

Easy enough, right? Now let's complicate it.

Multiple Entries

Each unique combination of access rules needs its own line in sudoers. It's perfectly legal to use multiple entries like this:

```
mike www=/sbin/reboot
```

```
mike www=/sbin/dump
```

This quickly gets cumbersome, though. If you have multiple similar rules, separate individual parts with commas.

```
mike,pete www=/sbin/reboot, /sbin/dump
```

The users **mike** and **pete** can run the reboot and dump commands on the host **www**.

While you can list multiple commands and users in a single rule, you must use different rules for different access levels.

```
thea ALL=ALL
mike,pete www=/sbin/reboot, /sbin/dump
```

The first rule declares that system owner **thea** can run any command on any host. She has graciously allowed minions **mike** and **pete** to run two commands on the host **www**.

Permitting Commands as Other Users

Some applications, usually databases or commercial Java programs, must be run by specific users to work correctly. Sudo lets you run commands as a user other than root, if the sudoers

policy permits it. List the user name in parentheses before the command.

```
kate beefy = (oracle) ALL
```

The user **kate** can run any commands on the server **beefy**, but only as the user **oracle**. She can fully manage the database, but has no special privileges otherwise.

Users with access to specific user accounts can also have separate access to root-level privileges.

```
mike beefy = (oracle) ALL
mike beefy = /sbin/mount, /sbin/umount
```

mike can mount and unmount disks, as well as manage the Oracle database.

Long Rules

Once you list multiple commands by full path, multiple users, and multiple machines in a single rule, individual sudoers lines can get really long. End a line with a backslash to indicate that the rule continues on the next line.

```
kent,mike,pete beefy,www,dns,mail = /sbin/mount, /sbin/umount, \
    /sbin/reboot, /sbin/fsck
```

Whitespace and additional lines make rules easier to manage. Use them liberally.

Edges

Here are a couple last points about sudoers.

Sudo processes rules in order, and the last matching rule wins. If two rules conflict, the last matching rule wins. We'll see how this comes into play as we build complex sudoers rules.

The exclamation point (!) is the negation operator. It's used to exclude one item from a list. You could say that a rule applies to everything except a specific user, host, or command. It also turns off options. Remember that the exclamation point means "not." The rest of this book has many examples.

Finally, a sudoers file must always end in a blank line. If visudo indicates an error on the last line, but the syntax all looks correct, verify that you have a blank line at the end of your policy.

Now that you have a basic grasp of sudo and sudoers, let's create our own sudoers file and test it with sudo.

Chapter 3: Editing and Testing Sudoers

If sudo cannot parse */etc/sudoers*, it will not run. If you rely on sudo to get root privileges on your server and you break sudo, you lock yourself out of the server's privileged commands. Fixing sudoers is a privileged command. This is a bad situation. Don't put yourself here. Sudoers must contain valid syntax. Sudo includes a tool specifically for editing sudoers, visudo.

Visudo protects you from obvious sudoers problems. It locks */etc/sudoers* so that only one person at a time can edit it. It opens a copy of the file in your text editor. When you save the file, visudo parses it and checks the sudo grammar. If your new sudoers file is syntactically valid, visudo copies the new file to */etc/sudoers*.

Remember that "syntactically valid" is not the same as "does what you want."

Visudo defaults to using the vi editor. While all sysadmins must have a passing familiarity with vi[2], that doesn't mean you need to

do everything with it. Visudo respects the $EDITOR environment variable, so you can use your preferred text editor.

Set your preferred editor, and we'll go on to editing sudoers.

Creating /etc/sudoers

While most operating systems include a sample or default sudoers file with lots of examples, you're here to learn. Learning means making your sudoers policy from scratch, just like a cake but not as delicious. Move the default sudoers file somewhere so you can use it as a reference. When you run visudo, it creates a new file.

```
# visudo
```

Create a very simple sudoers file, giving your account full privileges to the server. Here Thea gives herself unlimited access via sudo.

```
thea ALL = ALL
```

Save the file and exit. With a simple rule like this, permitting one user full access to the machine, your text editor should exit cleanly and visudo should install the rules.

Now, as a learning exercise, break sudoers. (Ubuntu and Apple users, you *do* have a root password, right?) Run `visudo` and pound keys to create a line of garbage on the bottom of the file. Save and

exit. You'll see:

```
# visudo
>>> /usr/local/etc/sudoers: syntax error near line 3 <<<
What now?
```

If you press e, visudo returns you to the text editor to fix your problem. Go to the line specified and see what's going on. Remove the garbage, and visudo will let you exit the text editor and install the policy.

To throw away your changes and retain the old sudoers policy, press x. An old working sudoers is better than the new broken one. I did this more than once while learning sudo, so don't let it worry you at this stage.

If you press Q, you install the broken file as */etc/sudoers*. When sudo cannot parse */etc/sudoers*, it immediately exits. Pressing Q tells visudo to break sudo until you log in as root and fix it. Do not press this button. You won't like it.

If you forget these keys, entering a question mark prompts visudo to print out your options.

Remember that a valid sudoers file is not the same as a useful sudoers file. A blank sudoers, denying all privileges to everyone, is perfectly valid and very quick to parse. Visudo also accepts a sudoers file where every rule specifies users and commands not on the system, or a server other than the local system.

When you're creating the sudoers file for your network, I strongly recommend that the last rule gives your account the right to run visudo. If everything else fails, you can fix the rules.

```
thea ALL = /usr/sbin/visudo
```

Remember that sudo processes rules in order, and the last matching rule wins. Put your emergency rescue rule at the very end of the file.

Testing sudoers

You've written your first sudoers security policy. At the moment, you can read it pretty easily – it only has two lines: your full access entry and your emergency visudo entry. But when your policy gets more complicated, how can you tell what a user can access?

Users can use sudo's –l flag to list their privileges.

```
$ sudo -l
Password:
User thea may run the following commands on this host:
    (root) ALL
    (root) /usr/sbin/visudo
$
```

When Thea enters her password, she sees what commands she can run. This output might look a little odd, but it should also look a little familiar. It's a pretty standard sudoers entry, with the user and host removed. Remember, if you don't specify a user in sudoers, sudo runs the command as **root**. This might be a little more obvious with a slightly more complicated example:

```
$ sudo -l
```

```
User thea may run the following commands on this host:
    (root) ALL
    (oracle) ALL
    (root) /usr/sbin/visudo
```

Thea can run all commands as **root**, all commands as the user **oracle**, and `visudo` as root.

That's fine for a user to check their privileges, but what about the system administrator? How can you be sure that your sudoers policy works the way you think it should? Use the `–U` flag along with `–l` to specify a user.

```
# sudo -U mike -l
User mike is not allowed to run sudo on www.
```

Only root and users that can run ALL commands on the current host can use –U. With my unprivileged user account, I can only check my own access. Sudo sees that Thea has the magic ALL attached to her security policy, so she can view my access. Otherwise she'd have to run `sudo –u mike sudo -l`, which is kind of daft.

We'll use `sudo –l` throughout this book to see how complicated sudoers policies expand into user-visible rules. I recommend using

-U after a change to verify the user's access before telling him that the access he requested is available.

Chapter 4: Lists and Aliases

Writing a sudoers policy is simple. You just write down who can run what on which machine. What could be easier? Now repeat that for five hundred users. Make sure users with a common function have identical security rules. And those Oracle database administrators? You must include every single command each administrator needs to run as a separate user for each and every one of them.

If you had to write all this out in sudoers, you'd just spray-paint the root password on the wall of the break room instead.

To make things more complicated, Unix-like systems get information from a whole bunch of sources. Some of them aren't even vaguely Unixy. If a server is attached to an Active Directory or NIS domain, you might need to use that information in your security policy. Perhaps you want a rule that "all users in the Domain Admins group can mount CIFS shares." You need to know

how to draw this information into your sudoers policy.

Sudoers offers aliases to condense and simplify security policies. An alias is a predefined list of items that you can use in sudoers rules. You can use aliases anywhere you use a username, host, or command. Changing an alias is a simple, effective, and guaranteed consistent way to make changes in complex sudoers files.

But before we get into any of that, let's look at wildcards.

Wildcards

A wildcard is a special symbol that can match different types of characters. Sudoers lets you use wildcards to match hosts, filesystem paths, and command-line arguments. Sudoers wildcards look a lot like shell or Perl regular expressions, but aren't. Wildcards are built on the operating system's glob and fnmatch functions. If your operating system's glob and fnmatch functions support character classes, you can use classes in wildcards. If you don't know what character classes are, don't worry about it.

Matching Numbers and Characters

Suppose your network has several Domain Name Service servers, all with hostnames like **dns1**, **dns2**, **dns3**, and so on. You probably wouldn't give a non-DNS server a name starting with those characters. Your DNS administrator needs full access to these servers, so you could use a wildcard in the host definition.

```
fred dns? = ALL
```

The question mark (?) matches any single character. This sudoers rule applies to any host **dns0** through **dns9**. It also matches **dnsA** through **dnsz**. Maybe you only have DNS servers 1 through 4, don't foresee any expansion, and don't want to automatically give privileged access to your regular DNS admins on any new DNS servers that appear.

```
fred dns[1-4] = ALL
```

By specifying a range of characters in brackets, you restrict the match.

You can use a range of letters in brackets.

```
pete www[a-z] = ALL
```

Pete can run any command on the servers **wwwa** through **wwwz**. Not many people use letters this way, but it's an option. You can also use capital letters, and the range A-z matches all capital and lower case letters.

```
pete www[A-z] = ALL
```

If you want to match multiple characters of a type, append an asterisk.

```
fred dns[0-9]* = ALL
```

If you eventually have the server **dns9183**, Fred can manage it. He will be very tired by then, I'm sure, so hopefully you'll use a user alias to get him some help.

Matching Everything

The asterisk character, more generally, matches any number of characters or none at all. It matches everything, with some deliberate exceptions. If Thea needs Pete to manage a server's core functions, she could give him a rule like this:

```
pete ALL = /sbin/*, /usr/sbin/*, /usr/local/sbin/*
```

Pete can run any command in any of the common sbin directories. Visudo is probably in one of those directories, so Pete can change his own privileges. Thea needs to learn the fine points of access control, or maybe move visudo to a private directory.

When used for commands, the asterisk does not match the slash character used to separate directories. If you want a user to have access to all the programs in a subdirectory, you must explicitly

specify that subdirectory.

```
pete ALL = /usr/bin/*, /usr/bin/X11/*
```

When used for command-line arguments, however, the asterisk does match the slash. Commands might include slashes in arguments, after all. They might include whitespace, any text strings, or who knows what.[3]

This means sysadmins need to take care using wildcards for command-line arguments. It's hard to beat the textbook example of a dangerous wildcard rule:

```
pete ALL = /bin/cat /var/log/messages*
```

Pete can see the contents of /var/log/messages, as well as the rotated logs such as /var/log/messages.1. That seems harmless enough. But wildcards match any number of characters, so Pete could run a command like this:

```
$ sudo cat /var/log/messages /etc/shadow
```

This surely isn't what the system owner meant.

It's pretty easy to work around this. The question mark matches

a single character.

```
pete ALL = /bin/cat /var/log/messages, \
    /bin/cat /var/log/messages??
```

Or Thea could use a range of numbers.

```
pete ALL = /bin/cat /var/log/messages, \
    /bin/cat /var/log/messages.[0-9]
```

Narrower number ranges work, of course.

Matching Specific Characters

Sometimes you must match select characters, rather than a range. You might need to match any of the characters `A`, `c`, or `q`. There's no way to express these as a range, but you can match specific characters in square brackets.

```
pete ALL = /opt/bin/program –[Acq]
```

This pattern matches a single character specified within the brackets, allowing you to safely permit a user access to specific command-line arguments.

The characters `*`, `?`, `[`, and `]` have specific meanings in sudoers. If you need to match one of these characters, put a backslash before

it. Here we allow the arguments [and].

```
carl ALL = /opt/bin/program –[\[\]]
```

You can now permit any combination of arguments you desire.

Blocking Everything

Maybe you specifically want to forbid using any arguments at all. Two double quotes with no space between them tell sudoers to only match the empty string.

```
dirk ALL = /opt/program ""
```

Dirk can run the program specified only if he doesn't give any arguments.

Wildcards are especially useful combined with aliases.

Aliases

An alias is a named list of similar items. You can use aliases to refer to the user running the command, the hosts sudo is run on, the user the command is run as, or the commands being run. As a simple example, let's make an alias that includes the commands for backing up and restoring Unix-like systems using traditional dump.

```
Cmnd_Alias BACKUP = /sbin/dump, /sbin/restore, /usr/bin/mt
```

A user who can run these commands can create and deploy backups. Who has this thankless job?

```
mike ALL = BACKUP
```

Lucky me.

For one user, an alias might not seem like much of an advantage. If you have several backup operators, however, you can create a alias for their usernames. Here I create the TAPEMONKEYS alias for the people who manage backups.

```
User_Alias TAPEMONKEYS = mike, pete, hank
```

When you combine these aliases, you can write a sudoers rule like this:

```
TAPEMONKEYS ALL = BACKUP
```

Two alias declarations and one rule replace a much longer rule. You could write the exact same rule without aliases.

```
mike,pete,hank ALL = /sbin/dump, /sbin/restore, /usr/bin/mt
```

This is longer and more difficult to read. When you add commands or users, it grows longer still. And successful tape monkeys will pick up more duties, lengthening the command list.

Using aliases makes personnel and task changes instantly percolate throughout sudoers. There's no risk of dozens of cut-and-paste changes numbing your brain.

Alias names can only include capital letters, numbers, and underscores. The name must begin with a capital letter. CUSTOMERS is a valid alias name, but _CUSTOMERS and 2CUSTOMERS are not. You must define aliases before using them, so people normally put all aliases at the top of sudoers.

Now let's look at the four types of data found in sudoers, how to extend them, and how to use them in aliases.

User Lists and Aliases

Remember in Chapter 1 when I told you that every sudoers rule started with a username? Yeah, well… that's not exactly correct. Strictly speaking, each rule starts with a *list* of users. A username is the most common type of entry on this list, but there are more. There are many more.

The usernames sudoers recognizes aren't necessarily usernames from */etc/passwd*. My organization manages user accounts via LDAP, and sudoers recognizes LDAP usernames exactly like local usernames. But my particular LDAP configuration restricts usernames so they look like local ones. You might need to pull in information from Microsoft Active Directory, or */etc/group*, or a user alias, or some obtuse directory system only used by three New Guinea tribesmen and your cutting-edge organization.

Sudoers recognizes seven types of user lists.

Operating System Groups

Sudoers accepts groups from the operating system. Give the group name with a percent sign (%) in front of it. I could create the */etc/groups* entry **dba**, add my database administrators to it, and reference it in sudoers.

```
%dba db1 = (oracle) /opt/oracle/bin/*
```

Everyone in the dba group can run all the commands in the directory */opt/oracle/bin*, as **oracle**, on the server **db1**.

Some operating systems have a system group for users who can become root (**admin** on Ubuntu) or who may use the root password (**wheel** on BSD-based systems). The default sudoers policy has an example of giving these users unlimited system access.

```
%wheel ALL = (ALL) ALL
```

The people in this group don't get any additional access through this rule – members of **wheel** can already use `su` to become root. But this lets people acclimate to using sudo in their day-to-day work.

Remember, use the `id` command to see which groups your account is a member of.

User ID

You can use user ID numbers in sudoers by putting a hash mark (#) before them.

```
#10000 ALL = /sbin/reboot
```

Any account with the UID 10000 can reboot any machine via sudo. I don't know why you would want this user to run around rebooting everything, but I've seen configurations more bizarre than this.

If you have multiple user accounts with identical user IDs, this rule applies to all of those user accounts.

Group ID

If you don't want to use group names, use group ID numbers prefaced by %#. On a traditional BSD system, wheel is group 0.

```
%#0 ALL = ALL
```

If your user name service is flaky, you might want to go this route. I recommend you fix the name service instead, but you might not control that.

As with user IDs, if you have multiple groups with the same GID, this rule applies to both equally.

Netgroup

If you're managing your systems via NIS, your next step should be to stop using NIS. But until you get to that point, you can reference netgroups in sudoers rules by starting them with a plus sign (+).

```
+webmasters ALL = /opt/apache22/bin/*, /opt/apache22/sbin/*
```

Your webmaster team can run any of the programs in the two specified Apache directories.

Non-Unix group

If your version of sudo has the necessary plugins or additional code to support checking groups against information sources beyond the norms of Unix-like systems, you can reference those in sudoers. Preface them with %:.

```
%:Admins ALL = ALL
```

Many non-Unix directory services use spaces or non-ASCII

characters in group names. These characters must be escaped somehow. Escaping special characters is a pain, so enclose the entire group name (including the leading %:) in double quotes.

```
"%:Domain Admins" ALL = ALL
```

When in doubt about non-Unix groups, use double quotes.

When you run `id` to see which groups your account belongs to, non-Unix groups appear in the output after the standard Unix groups.

Non-Unix Group ID

So you've attached your system to a non-Unix directory and you want to use the number of those foreign groups rather than the names? No problem. Put `%:#` before the group number. Yes, that's a percent sign, a colon, and a hash mark.

```
%:#87119301 ALL = ALL
```

If you find yourself needing to do this, however, I suggest that you step back and reconsider how you're using your directory service.

User Aliases

Your list of usernames can include a user alias, so we'd better discuss those. A user alias is a list of system users. All user alias definitions start with the string User_Alias.

```
User_Alias SYSADMINS = thea
User_Alias MINIONS = mike, pete, hank, dirk
```

Here, the user alias SYSADMINS contains one user, **thea**. In the event that the organization gets another full systems administrator, adding their username to the alias will give the new person the same rights as Thea.

The user alias MINIONS contains four users. When Thea uses this alias in a sudoers rule, it affects all four minions identically. Other rules might alter an individual minion's access, of course.

You can use any type of usernames in a user alias.

```
User_Alias WHINERS = "%:Domain Users", %operator, MINIONS
```

Remember, alias names can only have capital letters, numbers, and underscores. The alias name must start with a capital letter.

Host Lists and Aliases

The hosts entries in sudoers accepts values other than pure host names. But let's talk about those pure host names first.

Sudo determines the name of the local host by running hostname. It does not rely on DNS, */etc/hosts*, LDAP, or any other name directory. The traditional host name **localhost** doesn't work in a rule unless that's what hostname returns. (You can change this behavior with the fqdn option, which we'll examine in Chapter 10.) This means that your hostnames in sudoers must match the hostname set on the local machine. Change the hostname and sudo breaks. If hostname returns a fully qualified domain name (e.g., **www.michaelwlucas.com** instead of **www**), then sudoers only needs the machine name, not the full domain name.

In addition to using the local host name, sudoers can accept a variety of IP addresses and netgroups.

IP Addresses

Sudo can differentiate between host names and IP addresses, so you don't need to put any special markers in front of an IP address.

```
mike 192.0.2.1 = ALL
```

Sudo checks all of the machine's real network interfaces for IP addresses. It also checks interfaces attached to real interfaces, such as VLAN interfaces and bridges. It ignores logical interfaces such as the loopback.

You can also use networks in sudoers, specifying netmasks either in dotted-quad (192.0.2.0/255.255.255.128) or Classless Inter-Domain Routing (CIDR) format (192.0.2.0/24). If any interface on the machine is in that network, the sudoers rule applies.

```
pete 192.0.2.0/24 = ALL
mike 198.51.100.0/255.255.255.0 = /etc/rc.d/named *
```

For machines with multiple interfaces on different networks, remember that sudo uses the last matching rule. If the rules for two networks conflict, the last rule wins.

Netgroups

YP/NIS sites can refer to netgroups in sudoers by putting a + in front of the name.

```
carl +db = ALL
```

For most of us, however, the way to refer to groups of hosts will be with host aliases.

Host Aliases

A host alias is a named list of hosts. Indicate a host alias with the string Host_Alias. A host alias can include any variation of hostname recognized by sudo.

```
Host_Alias WWW = www[1-3]
```

You can include one host alias in another.

```
Host_Alias DMZ = 192.0.2.0/24, 198.51.100.0/255.255.255.0, WWW
```

Like user aliases, host alias names must contain only capital letters, numbers, and underscores, and must start with a capital letter. You can then use this alias in a sudoers rule.

```
mike DMZ = all
```

Now I have full privileges on the hosts in the DMZ group.

RunAs Lists and Aliases

You can grant a user permission to run a command as another user by putting the target username in parentheses before the command. We saw how to do this earlier:

```
chris beefy = (oracle) ALL
```

Chris can run any commands on the host **beefy** as the user **oracle.** These are called *RunAs privileges*.

RunAs Lists

Like usernames, RunAs users are lists. Suppose you have multiple database platforms – Oracle, MySQL, and Postgres. Your database team needs access to run commands on any host as the database user. Any type of username that's valid in a list of users is valid in a RunAs statement.

```
carl ALL = (oracle, postgres, mysql) ALL
```

Database administrator Carl can run any command on any server, so long as he runs it as one of the database user programs.

If you have non-Unix-style users who can run commands, you

can write sudoers rules that include them.

```
pete ALL = ("%:Domain Users", %operator, lpd) ALL
```

You can also let a user run a command as a member of a group, rather than as a specific user. Standard Unix convention is to specify file ownership with a username, a colon, and a group name. To write a rule that permits running a command as a group member, skip the username. You might have log files that are only visible to members of the group **staff**.

```
%helpdesk ALL = (:staff) cat /var/log/secure
```

Helpdesk staff can run this command as if they were in the group **staff**.

RunAs Aliases

You're probably getting the hang of this by now, but to be complete let's talk about RunAs Aliases. A RunAs alias lets you group users needed to run commands. The name of a RunAs alias can only include capital letters, numbers, and underscores, and must begin with a capital letter.

```
Runas_Alias DB_USERS = oracle, postgres, mysql
```

You can use the string DB_USERS anywhere you'd want to use a list of usernames.

```
carl DB = (DB_USERS) ALL
```

We now have a single, readable rule that lets Carl run anything as a database user, on any server in the DB alias. If Carl gets any help in database administration, the system owner can replace Carl's name with, say, a DB_ADMINS alias.

Command Lists and Aliases

In some ways, lists of commands are the simplest lists. A command can either be a path with a wildcard (*/sbin/**) or a full command name (*/sbin/dump*). You can put these commands in lists, as we've already seen.

```
mike ALL = /sbin/dump, /sbin/restore, /usr/bin/mt
```

There's no way to pull in non-Unix commands. What's on the filesystem is what you have to work with.

Command Aliases

Command aliases are lists of commands assigned a name, labeled with Cmnd_Alias. The rules for command alias names are exactly the same as other aliases. Command aliases can include other command aliases.

```
Cmnd_Alias HELPDESK = /usr/bin/passwd, BACKUP
```

You can use a command alias anywhere you'd use a command.

Command Tags

You can use *tags* before a command list or command alias. A tag is a flag that changes how the command runs. I'll show exactly what the ten tags do in more appropriate sections of the book, but you should recognize a tag when you see it. A tag appears before the command list, separated from the commands by a colon.

```
mike ALL = NOEXEC: ALL
```

Tag names are all capitals, without any numbers or symbols. A tag affects all the commands in the list following the tag. We'll use the NOEXEC tag in Chapter 6, so don't worry about what it means right now.

Excess Rules

Some rules are more generous than they need to be. Let's reconsider Carl's database access.

```
carl ALL = (oracle, postgres, mysql) ALL
```

Carl can run commands as the three database users on all computers in the organization. He doesn't need this access on all the machines, however. Most machines have only one database server or client installed on them. You see very few systems running both MySQL and Postgres.

In many environments, this extra access probably doesn't matter.

If Carl tries to run a command as **oracle** on a system running PostgreSQL, the command will fail.

```
$ sudo -u oracle sqlplus
sudo: unknown user: oracle
sudo: unable to initialize policy plugin
```

If the user exists, thanks to the wonders of LDAP, but there's no software, the command will fail. If the software exists, but isn't

configured, the command will fail. If the software is configured and the command fails, the database probably isn't running. And if Carl tries to configure Oracle on the PostgreSQL server, senior sysadmin Thea needs to have sharp words with him. Probably involving a tire iron.

When you write complicated policies, you will need to decide how much work you're willing to do to eliminate this excess access. Is Carl's ability to configure PostgreSQL on the Oracle server a risk? If it is, eliminate it.

Negation in Lists

Remember the ! character I brought up back in Chapter 2? We can use the negation character to exclude items from a list.

```
User_Alias NOTSCUM = %wheel, !mike
NOTSCUM ALL = ALL
```

The members of group wheel, with one exception, get full access to the system. Thea says that when I tell her what I did with her comfy chair, I might get my access back.[4]

Negation is very powerful for host, user, and Run As aliases. It is not only not useful for command aliases, it is actively harmful. Lists of commands include either the full path to specific commands, or a directory with a wildcard.

You'd think negation would be effective for command lists. But users can copy files. They can create links to files. They can find a way to access a file through a variety of paths. To see why this is a problem, here's an alias for the commands useful to become root.

```
Cmnd_Alias BECOME_ROOT = /bin/sh, /bin/bash, /bin/tcsh, /usr/bin/su
```

Here's a sudoers rule that excludes those commands.

```
%wheel ALL = ALL, !BECOME_ROOT
```

This seems to work. If I try to run a forbidden command, sudo tells me I'm not allowed and logs the error. Being an annoyingly clever user, though, I try the following:

```
$ cp /bin/sh /tmp/mycommand
$ sudo /tmp/mycommand
# id
uid=0(root) gid=0(wheel) groups=0(wheel),5(operator)
```

Oops. The sysadmin excluded */bin/sh*, but not the copy of */bin/sh* installed as */tmp/mycommand*. And certainly not the copy of zsh that I compiled myself and installed in my home directory.

You cannot use exclusions to remove commands from a list. There is no way to exclude commands securely or safely. The sudo authors have documented this extensively, have begged people not to do it, and *still* sysadmins all over the world insist on doing this. Nothing screams "I don't read the instructions!" like using exclusions in sudoer command lists. Exclude users. Exclude

machines. Even exclude Run As aliases. But don't exclude commands.

Aliases in Sudo

A user who checks his privileges with sudo –l will see the expanded aliases, not the aliase names or their definitions.

```
$ sudo -l
Password:
User mike may run the following commands on this host:
    (root) ALL, !/bin/sh, /bin/bash, /bin/tcsh, /usr/bin/su
```

I don't see the BECOME_ROOT alias, so I don't know how Thea wrote this policy. I do see *how* to get root on this machine, without Thea being any wiser. Because a sysadmin who doesn't configure sudo correctly certainly isn't reviewing the logs either (see Chapter 12).

Aliases are a simple way to rationalize and simplify your sudoer policy. Now let's see how to change the core of how sudo behaves through options and defaults.

Chapter 5: Options and Defaults

Sudo's standard behaviors accommodate the most common use cases. The interesting thing about the most common use case, however, is how uncommon it is. You can change most of sudo's core behavior by setting various options in sudoers. These options can be set as global defaults or attached to specific rules, hosts, users, or commands.

Set defaults with `Defaults` statements. A sudoers policy can have multiple `Defaults` statements. If multiple `Defaults` statements conflict, the last matching one applies. We'll see lots of sample `Defaults` statements throughout this chapter.

Most options that affect specific sudo functions have their own chapter, and are discussed in that chapter. That is, we cover environment-affecting options in Chapter 8 and logging options in Chapter 12. This chapter covers how to use options in general, both for specific groups and as defaults. We'll start by using options in

Defaults statements.

Option Types

Options can be either boolean, integers, integers or lists usable in boolean context, or strings.

Boolean Options

Some options affect sudo with their mere presence. They're toggle switches, turning behaviors on and off. Some boolean options are on by default, even when they don't appear in sudoers. Deactivate them by putting an exclamation point before them.

For many years, when a user typed the wrong password, sudo responded by insulting them. The sudo developers changed this a while back, apparently in an effort to make sudo seem more professional or enterprise-friendly. Insulting users is a sysadmin's prerogative, however, and automating insults demonstrates sysadmin competence. Put the insults option in sudoers to make sudo insult users who can't type their password.

```
Defaults insults
```

When the user types the wrong password, he'll receive motivational commentary in addition to a password prompt.

```
$ sudo -l
Password:
Sorry about this, I know it's a bit silly.
Password:
stty: unknown mode: doofus
Password:
Harm can come to a young lad like that!
sudo: 3 incorrect password attempts
```

If your sudo installation insults users by default, you can disable the insults by disabling the option.

```
Defaults !insults
```

Users now get the boring "Sorry, try again" message.

Some operating system packagers deliberately remove this option from their version of sudo. If yours does this, I recommend complaining bitterly until they see the error of their ways.

Integer Options

Some options take a number as an argument. Use an equal sign to

separate the argument from the option name. These options set a limit for this sudo option.

Common wisdom on passwords is that they should include mixed-case letters, numbers, and assorted symbols. Oh, and they should be long. This combines to make them hard to type, especially when the password isn't visible as you type it. Your users might need more than three tries to type their password correctly. Here, Thea lets users try to type their password five times before kicking them out of sudo and logging an error.

```
Defaults insults, passwd_tries=5
```

Here we combine two options in one Defaults statement, separated by a comma. You can use as many options on a line as you want, but I recommend grouping them by function.

Integers usable in Boolean Context

If an integer option sets a limit on sudo's behavior, these options let you disable a feature by setting the limit to zero. Do you remember that sudo caches the fact that you have authenticated for

five minutes? You can change the number of minutes it remembers.

```
Defaults insults, timestamp_timeout=10
```

The longer sudo caches the authentication, however, the greater the risk that the user will walk away from a privileged terminal session. Many users don't lock their workstations when they leave their desk. Using a longer timeout increases the odds of a security problem.

The way around this, of course, is to completely disable the authentication timeout. Require the user to enter a password every time they run sudo. By setting the timeout to zero, you entirely disable authentication caching.

```
Defaults insults, timestamp_timeout = 0
```

Depending on your environment, and what commands people use for sudo, disabling the authentication timer might be too harsh. But this makes sense if you're using strong authentication methods, as we'll see in Chapter 13.

String Options

Some options need arguments like text or a path to a file.

When a user mistypes his password, there's a middle ground between insulting the user and offering a bland "Sorry, try again." You can use a custom message by setting the badpass_message option.

Defaults badpass_message="Wrong password. I have noted your incompetence. Try again!"

When the user mistypes his password, sudo displays the custom message. I put the message in quotes because it has special characters, like spaces and the exclamation point. Options that take a file path as an argument don't need the quotes.

Setting Options for Specific Uses

Options aren't just global defaults. You can set options on an individual basis, so that they only affect certain users, commands, or specific machines.

Per-User Defaults

Certain users should get different default settings than others. Perhaps you need to set different authentication timeouts for some users, or a different password prompt, or some whiner complained that the system insulted him. You can change the defaults for specific users. Use the keyword `Defaults`, a colon, the user or a list of users, and the option.

The first time you run `sudo` on any machine, it prints a short lecture reminding you to be careful. Most users need the reminder. But system administrators are continuously mindful of their responsibilities and are painfully aware of the damage they can do with a misplaced keystroke.[5] They don't need reminding, and

once you've seen the lecture hundreds of times, it only annoys you. Here Thea disables the lecture option for herself.

```
Defaults:thea !lecture
```

She could also disable the lecture for everyone allowed to use the root password.

```
Defaults:%wheel !lecture
```

The people who have root privileges will now be very slightly less annoyed. Which can only be good.

Per-Host Defaults

To override sudoers defaults on a per-host basis, use Defaults, an at symbol (@), the list of hosts or host alias, then the option. Anything that can be in a hosts alias can appear here.

```
Defaults lecture
Defaults@TESTHOSTS !lecture
Defaults@PRODUCTION lecture=always
```

Here we have two host aliases. In the test environment, users are not lectured. In production, however, every time sudo asks for their password it also lectures them. I recommend reserving this last

feature for truly troublesome users.

Per-Command Defaults

To set per-command or command alias defaults, use Defaults and an exclamation point.

Perhaps some users can be trusted, most of the time. But maybe a specific user has difficulty with a certain command. Or maybe a certain problem has happened once too often.

```
Defaults !lecture
Defaults!/sbin/fdisk lecture=always, \
    lecture_file=/etc/disklabel-lecture
```

The lecture_file option lets the sysadmin write a custom lecture message. In this case, */etc/ disklabel-lecture* contains a text message to replace the standard lecture.

```
If you relabel a vital disk again, Thea will
leave the tatters of your still-living body
in the break room as a warning to others.
```

The lecture appears only if the user must enter their password, but that's better than nothing. To make the lecture appear every

time he uses this command, require the user to enter a password every time.

```
Defaults!/sbin/fdisk lecture=always, \
    lecture_file=/etc/disklabel-lecture, \
    timestamp_timeout=0
```

By setting timestamp_timeout to zero for this specific command, Thea removes the timeout on authentication. Whenever a user runs fdisk, sudo displays the threat – er, *lecture* – and demands a password.

Tags can be defaults.

```
Defaults!ALL noexec
```

This default sets the NOEXEC tag set on all commands.

Per Run As Defaults

Lastly, to set a default for a Run As rule, use a right angle bracket between Defaults and the user list.

```
Defaults>operator lecture
```

Anyone who runs commands as **operator** (normally, the backup team) gets lectured.

Conflicting Defaults

Consider the following sudoers policy.

```
Defaults:mike insults
Defaults!/usr/bin/su !insults
mike ALL = /usr/bin/su
```

The first line says to insult me whenever I run sudo. The second line says that whenever someone runs su via sudo, don't insult them. The third line gives me the right to run `su`. The defaults conflict. What happens?

```
$ sudo su
Password:
Sorry, try again.
```

Sudo does not insult me.

Remember, sudoers policies work on a last match basis. The last matching Default statement says "don't insult su users." To insult me, reverse the order of the two Defaults statements.

Now that you know how to use options, we'll see them in play through the rest of this book.

Chapter 6: Shell Escapes, Editors, and Sudoers Policies

Unix-like operating systems and their software grow new features like moss grows on the Oregon Coast. They're everywhere. Many older but popular programs, such as the pagers `more` and `less` and the editor `vi`, let users run shell commands from within them. Try it yourself – view a file with `more`. While you're still looking at the file, enter an exclamation point and then a shell command such as `ls` or `ifconfig`. The command will run. You'll see the output, then `more` returns to the text it originally displayed.

Systems administrators who worked on dumb terminals or over SLIP connections desperately needed the ability to escape to a shell. You didn't want to leave a file just to verify if the IP address on your machine matched something in the file. Now that we can have umpteen SSH sessions open to a single machine, shell escapes aren't used so much.

Unless you use sudo. Then shell escapes become really awesome, in a bad way. Consider the following sudoers policy:

```
mike ALL = /usr/bin/more
```

I can use `more` to view files on any system. That's cool. I can look at, say, */var/log/auth.log* to see why a user's SSH connections fail. But I'm running `more` as root. That means any commands that I can get `more` to run, will run with root privileges. I run `sudo more` on a file, then enter:

```
!visudo
```

I'm in visudo, the sudoers editor! I can edit the policy to add a rule permitting me to run all commands on all machines, save, and exit. Then I quit `more` and check my privileges.

```
$ sudo -l
User mike may run the following commands on this host:
    (root) /usr/bin/more
    (root) ALL
```

If the senior sysadmin discovers this, she'll have my head on a platter. Again.

If a user has access to a limited subset of privileged commands,

you must ensure that he cannot bootstrap himself into greater access. Do this either through restricting the commands, or by prohibiting commands from running other commands.

Command Restrictions

One way to eliminate shell escapes is to verify that no permitted program includes shell escapes. This is hard – many programs have shell escapes, not just pagers and text editors. You could eliminate the pager issue by only allowing the users privileged access to cat(1), requiring them to dump the output to a pager.

$ **sudo cat auth.log | less**

This eliminates only shell escapes from pagers, however. To follow this method, you must carefully check the documentation of every permitted command for shell escapes. And not all documentation is complete.

Forbidding Commands from Executing Commands

Shell escapes aren't the only way to break out of a program. Many programs run other programs. We've already looked at visudo, which runs a text editor for you. On modern Unix-like operating systems, sudo can stop programs from executing other programs. Sudo uses the LD_LIBRARY_PRELOAD environment variable to disable program execution. Every modern BSD, Linux, and Unix-like operating system supports this variable, but check your system's documentation if you're uncertain.

The EXEC and NOEXEC tags control whether a command may execute further commands. EXEC, the unwritten default, permits execution of commands by other commands. NOEXEC forbids execution. Put the tag before the command in your sudoers rule.

```
mike ALL = NOEXEC: ALL
```

What does this do? Use `sudo more` to examine a file, and try a shell escape into `visudo`. Instead of getting into the visudo editor, `more` just prints a message like "done" or "exec failed." Why is it

done? It tried to run the command and failed.

The NOEXEC tag even disables running visudo via sudo.

```
$ sudo visudo
visudo: unable to run /usr/bin/vi: Permission denied
visudo: /usr/local/etc/sudoers.tmp unchanged
```

The visudo command tries to run a text editor. Visudo cannot run additional commands, so it fails.

A global NOEXEC tag is kind of harsh, though. Some commands legitimately spawn other processes to do tasks for them. For example, the `newaliases` command legitimately runs `sendmail`. I recommend using a global block, and then whitelisting specific commands.

```
mike ALL = NOEXEC: ALL, EXEC: /usr/bin/newaliases
```

The `newaliases` command is permitted to spawn new processes. A very savvy intruder could perhaps get newaliases to spawn a privileged shell, but that attack considerably raises the skill needed to penetrate your system.

A whitelist of permitted commands is a perfect application for a

command alias.

```
Defaults!ALL noexec
Cmnd_Alias MAYEXEC = /usr/bin/newaliases,/usr/local/sbin/visudo
mike ALL = ALL, EXEC: MAYEXEC
```

A user could run `sudo /bin/sh`, but that new shell won't be able to execute any commands other than those built into the shell.[6] The user could still damage the system, but doing so demands greater expertise. Many third-party sudo tutorials suggest specifically forbidding specific programs from executing other programs, much as they suggest excluding commands from a permitted list. Both solutions have the same problem. The only way to have true security through sudo is to explicitly enumerate the commands users may use.

Editing Files

Many editors offer shell escapes. But you need access to an editor to change certain critical files. You might try a sudoers policy like this.

```
mike ALL = NOEXEC: /usr/bin/vi /etc/ssh/sshd.conf
```

Would this give the ability to change the file, without shell escapes? Yes. But it has more general problems. First off, I am not using old-fashioned vi for day-to-day work. I prefer either Emacs or `ed` (if I must use a primitive editor, I want one that demonstrates that I'm an alpha geek). And I might have a legitimate need for an unprivileged shell escape while editing the file.

That's where sudoedit comes in. Sudoedit lets a user edit a privileged file without running an editor as root. When you run `sudoedit` on a file, sudo copies the target file to a temporary file, sets the permissions on the temporary file so you can edit it, and runs your editor on it. You edit the file with a normal, unprivileged text editor. When you exit the editor, sudoedit inspects the temporary

file. If the file has changed, it copies the temporary file to the target file.

Configuring Sudoedit

To configure editing permissions, use the sudoedit keyword and the full path to the target file.

```
%wheel ALL = sudoedit /etc/ssh/sshd_config
```

Users in the wheel group can edit the SSH server configuration file through sudo.

Using sudoedit

To edit a file, use the sudoedit command and the filename.

```
$ cd /etc/ssh
$ sudoedit sshd_config
```

A text editor opens. The user can make changes, save, and exit. Sudoedit puts their edited file in place of the original.

What editor does the user get? That depends on the user's environment. If the editor has a $SUDO_EDITOR environment

variable, that's used. Otherwise, sudoedit looks for $VISUAL or $EDITOR variables. If those don't exist, sudoedit looks for an editor option in sudoers. Sudoedit uses vi as a last resort. I encourage you to set an editor in sudoers, as vi is kind of boring.

```
Defaults editor=/bin/ed
```

Give the full path to the default editor. If a user can't use your editor and can't set his own, he shouldn't be editing the sudoers policy.

Writing Sudoers Policies

You now have all the pieces that make up a sudo policy. Everything else builds on what you've already learned. Let's discuss how to using these tools to build a sudoers policy.

In Chapter 4 I demonstrated how excluding commands from ALL lets people run arbitrary commands as root. In this chapter, I've demonstrated how shell escapes give people root access. While sudo logs all commands by default, it doesn't automatically log everything that happens. Programs like sudoreplay give more detailed logs but need special configuration (Chapter 12.) The natural question is: what good are the sudo tools if a user can avoid restrictions so easily?

If your users can run arbitrary commands as root, it's not the fault of the tool. The problem is that you've written your sudoers policy badly. Don't be too embarrassed – most people write poor sudoers policies. Many operating systems ship with a sudoers policy that permits all users in an administrative group unlimited

access. This policy means that your administrators can do anything *without even being logged*. A malicious intruder or administrator can hide an awful lot of damage behind a shell escape.

So, what to do?

The only way to write a secure sudoers policy is to deny commands by default. Use of the ALL keyword in a command gives people too many easy ways to gain unlimited privileged access. Users will work furiously to get around restrictions that they believe are in their way. Don't leave them a hole to squirm through.

Consider your sudoers policy like a firewall. Back in the 10baseT era, people ran firewalls that permitted all access and then blocked traffic to vulnerable services. On today's Internet, that's a sign of incompetence. Treat your sudoers policies the same way. Default permit sudoers rules make me proclaim "The 90s called, they'd like their security policy back."

The mere presence of the word ALL in the command portion of

a sudoers rule means that the user can get unrestricted root access regardless of any restrictions you might think you're placing on him. You cannot realistically enumerate badness in a sudoers policy any more than in a firewall; the only safe practice is to permit known necessary activity.

You can safely use ALL for users, Run As, and server lists. Unprivileged users can't change their username or a server's hostname, but they can change the full path to commands without trouble.

From this point on, I never use ALL in the command description except for specific examples of poor practice. To do otherwise is to invite abuse and intrusion. It's one thing to not be embarrassed by errors when you're starting out, but now you know better.

Chapter 7: Configuring sudo

Wait just a cotton-pickin' minute… isn't this whole book about configuring sudo? What have we been reading about, anyway?

We've been configuring sudo security policies in sudoers. The configuration of the sudo program itself depends on how sudo was built, and how the systems administrator changed the sudo client configuration via *sudo.conf*.

Sudo's Default Configuration

The sudo software suite as downloaded from the master web site ships with a default configuration, but your operating system packager has probably changed some of those settings. You can identify the actual defaults of your local install by running sudo –V.

```
$ sudo –V
Sudo version 1.8.7
Sudoers policy plugin version 1.8.7
Sudoers file grammar version 43
Sudoers I/O plugin version 1.8.7
```

Here a normal user has asked sudo for its configuration, and gets sudo's version number and a few basic facts about the configuration.

To really see what's inside your sudo install, use the –V flag as root.

```
# sudo -V
Sudo version 1.8.7
Configure options: --sysconfdir=/usr/local/etc --with-ignore-dot --with-tty-tickets --with-env-editor --with-logincap
```

```
…
Sudoers policy plugin version 1.8.7
Sudoers file grammar version 43

Sudoers path: /etc/sudoers
nsswitch path: /etc/nsswitch.conf
Authentication methods: 'pam'
Syslog facility if syslog is being used for logging: local2
Syslog priority to use when user authenticates successfully: notice
…
```

This goes on for over a hundred lines. You'll see how this sudo binary was configured to compile, where it looks for its files, how it authenticates, which environment variables it automatically purges and which it allows to pass unscathed, and more. Take a look at this output on your own sudo installation.

You can change some of these settings with entries in *sudo.conf*.

sudo.conf

You can configure the sudo program itself in */etc/sudo.conf*. Sudo usually runs just fine without any configuration file, but if you need to debug a problem or change basic behavior you need to understand *sudo.conf*. The file has four valid configuration types: *Plugin*, *Path*, *Set*, and *Debug*. Chapter 12 includes information on debugging sudo, so look there for details on the Debug flag. For each of the others I'll give one simple example of how sudo uses that type of configuration, but I'll refer to these types of settings in later chapters.

Plugins

A sudo plugin changes how sudo behaves at a fundamental level by replacing either the policy engine or the input/output system. You can use a plugin to replace */etc/sudoers* with your own security policy language – actually, sudo learns that sudoers exists because of the sudoers plugin. If you want to build a special logging system, use

an I/O plugin. Plugins are a new feature as of sudo 1.8, so the only free plugins that exist as I write this are the defaults. Commercial firms such as Quest One (http://www.quest.com) have already written sudoers and logging plugins, and others are sure to follow.

To use a plugin, give the Plugin keyword, the name of the plugin, and the name of the shared library. Here I explicitly configure the sudoers security policy and the sudo input/output logging module (Chapter 12).

```
Plugin sudoers_policy sudoers.so
Plugin sudoers_io sudoers.so
```

Sudo's shared libraries install in */usr/local/libexec/sudo* by default, but you can put an explicit path in *sudo.conf*. If you build sudo with a non-standard location, the build process sets the appropriate default directory. If you have a custom-built sudo plugin or something from a vendor, however, you might have to give the full path.

```
Plugin sudoers_policy /opt/custom/moderninsults.so
```

You should only need to explicitly define the full path if you're

writing sudo code and want to point at your specially built library.

You can only have one sudo policy engine at a time. If you use the Quest policy engine, you cannot also use sudoers. The point of having an external policy engine is that it can do things that sudoers can't. You can use multiple logging systems.

Paths

Sudo can use external programs and libraries for select functions. I'm using the noexec tag as an example, but we'll refer to the Path setting throughout this book as needed.

The NOEXEC tag uses a shared library to replace the system calls that execute programs with system calls of the same name that return errors. This tag relies on a shared library that includes the dummy functions. You should never need to use any noexec shared library other than the one included with sudo, but here's how you would set it.

```
Path noexec /usr/local/libexec/sudo/sudo_noexec.so
```

You'll normally use a path to do things like call an external

password program (see Chapter 8).

Set

Sudo has a few features controlled through Set commands. These are generally switches with predefined values such as *true* and *false*. I'll use core dumps as an example.

Sudo handles sensitive security information. It normally keeps that information in memory, and discards it as soon as possible. A core dump file from a crashed sudo process would contain all of that sensitive security information. Sudo therefore disables core dumps by default. If you want to enable coredumps, set disable_coredump to false.

```
Set disable_coredump false
```

This setting handles the sudo part of creating a core file, but most operating systems don't let setuid programs dump core. On FreeBSD, enable core dumps from setuid programs by setting the sysctl kern.sugid_coredump to 1. On OpenBSD, set the sysctl kern.nosuidcoredump to 0 to allow setuid programs to dump core. On

Linux, set the sysctls kern.suid_dumpable and fs.suid_dumpable to 2.

From here on out, I'll refer to making entries in *sudo.conf* and expect you to understand.

Chapter 8: User Environments versus Sudo

A user's shell environment might not be conducive to good system management. Environment variables exist to alter software behavior. Software running with elevated privileges needs to behave well, and environment variables which change that behavior can threaten your system. For that reason, sudo defaults to removing most of the user's environment before running any command.

If you're not sure what's in your environment, run the command `env`. You should see some familiar items in there, such as SHELL and PATH, but you will also see a bunch of less well-known variables like SHLVL or G_BROKEN_FILENAMES or EDOOFUS or whatever. Some of these are probably important. Many of them aren't. You might not even know how or where these variables get set. Purging the environment helps ensure that privileged commands run as they should.

Dangerous Environment Variables

How can environment variables be dangerous? Programs check environment variables for their settings – for example, shells use $HOME to identify the user's home directory. These environment variables are part of what makes a system Unix-like.

On the other hand, some programs use the environment variable LD_LIBRARY_PRELOAD to identify directories that contain additional shared libraries. But that directory might contain a version of libc that copies authentication credentials to a remote server. And there's a whole family of LD_ variables used on different operating systems. Shells like bash use $IFS to give the character that separates command-line arguments. Changing IFS to a carefully-chosen value can make processes do wildly unexpected things. If you lose your term paper because an incorrect environment variable made your text editor eat your files, that's annoying. If you use that same environment with a privileged command, you might lose more than your own files.

Programs can look for any environment variable. Commercial software often uses hundreds of environment variables to store arbitrary configuration data, much as Microsoft Windows uses the Registry. There is no master list of dangerous environment variables, as what is safe on one system can devastate another.

Sudo lets you carefully control your shell environment.

Execution Environment

Sudo doesn't just run a privileged command for you. It spins up an instance of a shell, runs the command, exits the shell, and returns control to the shell you ran sudo from. This is why commands like `sudo cd /opt/secret` don't work the way you might expect. Say your command prompt is in your home directory. You run the `cd` command. Sudo starts up a shell and changes into the desired directory. Then that shell exits. Your running shell instance is still in your home directory, while the shell instance in the desired directory no longer exists.

You want to see what's in that secret directory? Try `sudo ls /opt/secret`. You want to run a more complicated series of shell commands? Explicitly start a shell instance and write your commands as a quoted string.

```
$ sudo sh –c "cd /home ; du –d0 | sort –rnk 6"
```

Here I start a shell instance, gather the total size of all the directories in */home*, and sort them by size, largest first. The exact

specifics of this shell command don't matter; the point is that I had sudo run a list of shell commands via `sh –c`. You still need privileges to run `sh`.

Sudo bases the initial environment of the new shell instance on your environment, unless you tell it not to. You can tell sudo to establish this environment in three different ways: take your current environment and pass through selected environment variables, take your environment and strip out select environment variables, or abandon your environment and use the target user's environment. We'll cover each separately.

Whitelisting Environment Variables

By default, sudo removes all environment variables except $TERM, $PATH, $HOME, $MAIL, $SHELL, $LOGNAME, $USER, and $USERNAME. This means sudo runs commands in your preferred shell, with your regular path, and doesn't automatically dump created files in root's home directory. Sudo

also automatically removes any environment variable that begins with the characters `()`, as these can be interpreted as Bash functions. All well and good… until you need some other environment variable.

This is where the `env_keep` sudoers option comes in. `env_keep` lets the system owner define a list of environment variables that sudo should retain. For example, several environment variables control language and character set display options. If you're a native Russian speaker, you probably want commands that run under sudo to use your preferred character set.

```
Defaults env_keep += "LANG LANGUAGE LINGUAS LC_* _XKB_CHARSET"
```

Note the `+=` after the option name. This means "add the following to any existing list." If you use a plain equal sign, the option will overwrite the defaults. You'll get your character set, but lose your path, shell, and home directory. You could also use `-=` to subtract an environment variable from the list.

You can have as many `env_keep` statements as you need, and can

match them to specific user, machine, command, and RunAs lists. Perhaps administrators can keep their SSH environment variables, so they can copy privileged files across the network via SFTP.

```
Defaults:%wheel env_keep += "SSH_CLIENT SSH_CONNECTION \
    SSH_TTY SSH_AUTH_SOCK"
```

Or maybe you're stuck behind a proxy server, and everybody needs the proxy in their environment.

```
env_keep += "ftp_proxy FTP_PROXY http_proxy HTTP_PROXY"
```

You can pass any needed environment variable into the sudo environment.

Blacklisting Environment Variables

Leaving the user environment intact except for environment variables known to be dangerous is another example of enumerating badness. If you intend to shoot yourself in the foot, however, here's how to load the handgun.

The `env_reset` option tells sudo to remove all environment variables except a trusted few. It's set by default. To turn this off,

explicitly disable it in sudoers.

```
Defaults !env_reset
```

Even if you want to pass most environment variables unscathed, there's probably a few you need to strip from the environment. Use the `env_delete` option to remove an environment variable.

```
Defaults env_delete += "LD_LIBRARY_PRELOAD"
```

Users retain their entire environment, except for LD_LIBRARY_PRELOAD.

Running `sudo sh` would let the new shell instance read in a new copy of these variables from a configuration file, and you can certainly set them yourself inside the shell. But when you run an individual command, sudo will strip these variables from the environment.

Just like `env_keep`, `env_delete` lets you add environment variables to the deletion list based on groups, commands, and so on.

Allowing User Overrides

Some users, running some commands, might need to customize

their environment in ways the security policy can't anticipate. An application server might behave differently depending on the presence or absence of environment variables, and if the software changes quickly those values might need constant updating. Sudoers lets you write a security policy that says "Here are the standard environment settings, but let these specific users set their own environment variables for these specific commands."

Use the SETENV and NOSETENV tags on commands to let the user ask sudo to not alter his environment variables. The SETENV tag permits users to keep their environment on request. Here, Pete has a specific exception permitting him to control his environment on certain commands.

```
pete dbtest1 = (oracle) SETENV: /opt/oracle/bin/*
```

On the machine **dbtest1**, Pete can use his own environment when running Oracle commands as **oracle**. Oracle software is highly sensitive to environment variables. Pete can explore arbitrary configurations on the test server, and make a formal request for an

updated sudoers policy in production once he understands what he needs.

Pete must specifically ask sudo to not change his environment by using the –E flag.

```
$ sudo –E –u oracle /opt/oracle/bin/sqlplus
```

Without the –E flag, sudo will perform its standard environment stripping despite the presence of NOSETENV in sudoers.

Use the tag NOSETENV to override a previous SETENV.

```
pete dbtest1 = (oracle) SETENV: /opt/oracle/bin/*
pete dbtest1 = (oracle) NOSETENV: /opt/oracle/bin/gennttab
```

Pete can control his environment for all Oracle commands, except for gennttab. (Remember, sudo rules are last match.)

In addition to the SETENV tag, there's also a setenv option. Use it just like any other option.

```
Defaults:thea setenv
```

Thea can override her environment anywhere, provided she uses the –E flag with sudo. As the senior sysadmin she's already on the hook for system damage, and she needs the flexibility to

troubleshoot any possible problem. Giving herself the ability to override the environment on demand is a perfectly legitimate exception, especially as it only works at those times she specifically requests it.

Only give highly trusted users the ability to override environment variables, and then only in test environments. Remember, sudo policies aren't just to control users – they're also for limiting the damage malicious intruders can inflict on the system.

Target User Environment

I once sat in a meeting which boiled down to "The server runs fine unless Dave restarts it." The administrative solution was to fire Dave, but the technological solution was fixing how sudo managed Dave's environment. (Fortunately for Dave, the technological solution prevailed.)

In some cases you don't want to carry any environment

variables into your privileged environment. You don't even want your shell or home directory – instead, you need to run the command as the target user, in the target user's shell environment. Sudo lets you do that with the –i option.

By using sudo –i you simulate a new login as the target user, reading the target user's dotfiles such as *.login* and *.profile*, then running the requested command. Your original user environment is not retained in any way.

```
$ sudo –i /opt/apps/bin/start-server
```

In my experience having sudo initialize an environment as the target user is the best way to manage application servers highly dependent on their startup environment. Many Java server-side applications take their configuration from environment variables, and those variables might not be correct in your personal environment. By configuring that environment in a single account, you eliminate one threat to the application's stability.

Sudo Environment Defaults

Different releases of sudo might behave differently with regards to environment variables. I don't expect any of the default pass environment variables to change, but a future release of sudo might add new ones.

To learn about the environment-handling defaults on your version of sudo, run `sudo –V` as root. The output tells you how this system's particular sudo binary was built and how it treats different environment variables. You'll see three groups of variables: variables to sanity-check, variables to remove, and variables to preserve.

For sanity checking, sudo checks the listed variables for the characters `%` and `/`, removing them if present. Some environment variables affect your basic session – for example, a bad TERM variable can scramble commands as you type them. It's better to run a command without TERM set than run a command with a garbage terminal.

You'll see a list of "environment variables to remove." Sudo

does exactly that. You cannot override this list with env_keep; if you want these variables in the sudo environment, you must set them within the target user's account.

The list of environment variables to preserve is in addition to the list given earlier this chapter. You keep variables such as HOME and PATH, but also those shown by your specific sudo build.

Sudo-Specific Variables

A command run under sudo gets four sudo-specific environment variables: SUDO_COMMAND, SUDO_USER, SUDO_UID, and SUDO_GID. The SUDO_COMMAND variable is set to the exact command you ran under sudo to start this session. SUDO_USER gives your original username. SUDO_UID and SUDO_GID give your original user ID and primary group ID.

A program or script can check for the presence of these variables and behave differently if they're present or use them in

some way. You could use SUDO_USER in log messages, for example. "Yes, I was run by root, but really, I was run by **mike**. Blame him."

Environment Customization

A sudo policy can do more than just allow and disallow environment variables; it can explicitly set variables. Sudoers policies let you set the user's path, and you can also set arbitrary environment variables if needed.

Managing $PATH

One environment variable is a little trickier than most. Many intruders try to sabotage a user's $PATH, so that the user will run a bogus version of commands rather than the proper one. If a helpdesk flunky needs to reset a user's password, but he runs the program */tmp/.1234/hacker/passwd* rather than */usr/bin/passwd*, bad things will happen. Use the secure_path option to define your trusted path for sudo commands.

```
Defaults secure_path="/bin /usr/bin /sbin /usr/sbin"
```

Sudo tries to run the command using the secure path. If the command isn't in the secure path, it fails.

This affects commands run via sudo, but not shell instances started via sudo. If you start a full interactive shell, the shell reads the target user's *.profile* and other shell startup files as it initializes the environment. Secure paths help when running sudo like this:

```
$ sudo passwd mike
```

In this use case, secure_path makes sure that the passwd command being run is actually the system's passwd command and not an intruder's customized copy. It doesn't verify that the sudo command the user run is the proper one, however, so users still need to take care of their $PATH.

Adding Environment Variables

Sometimes you want to specifically set environment variables for a privileged user. Use the env_file option to give the full path to a file containing the new environment variables. One common situation is when you're behind a proxy server. You want users to always access the internet via your proxy? Add the environment variables to their environment.

```
Defaults env_file="/etc/sudoenv"
```

The environment file contains a standard list of variable assignments, like so.

```
FTP_PROXY=http://proxyhost:8080
ftp_proxy=http://proxyhost:8080
HTTP_PROXY=http:// proxyhost:8080
http_proxy=http:// proxyhost:8080
```

Sudo adds these environment variables before stripping out the environment, so list any added variables in an `env_keep` sudoers rule as well. This also means you override the user's own environment variables, so if a user has a different setting you've just replaced it.

Starting Shells with Sudo

Some people use sudo as a replacement for su. Essentially, they become root without using a password.

```
$ sudo su
```

I don't encourage this. Sudo logs which commands people use, but without additional configuration sudo doesn't log what happens inside a shell session. (We'll cover sudo logging in Chapter 12.) But since some of you do it anyway, let's discuss it.

The su command means "switch user." Running `su –` or `su –l` initializes a new shell just like using `sudo –i`. You get the target user's environment. Running plain `su` switches the user you're running as but retains most of your environment.

If you want to completely replace su with sudo, you could enable the `shell_noargs` option. With this option set, running sudo with no arguments gives you a root prompt.

```
Defaults:thea shell_noargs
```

When Thea runs sudo without any command-line arguments,

she's root.

```
$ sudo
Password:
#
```

You can simulate shell_noargs on the command line by using the –s flag.

```
$ sudo -s
Password:
#
```

If the user does not have permission to run root's shell, sudo denies access even if shell_noargs is present.

Another popular use of sudo is to run a shell, but retain your own environment.

```
$ sudo su –m
#
```

This leaves your shell unchanged and retains any environment variables your sudoers policy passes.

Which should you use? Ideally: none. If you must let users become another user via sudo, configure complete session logging

as per Chapter 12.

Sudo Without Terminals

Sometimes you want to run sudo without an attached terminal. You might want a right-click menu in your desktop manager that runs a program via sudo. This sudo program won't run in a terminal, however, so sudo can't ask you for your password. You need a way to get sudo your password.

Sudo can run an external program to prompt for the password. Use the askpass path in *sudo.conf* to tell sudo where to find this password program. The graphical password prompt software most likely to be found on any desktop system with sudo is OpenSSH's askpass, openssh-askpass.

```
Path askpass /usr/local/bin/openssh-askpass
```

When sudo needs a password and doesn't have a terminal to ask for one, it uses the askpass setting from *sudo.conf*.

Requiring a Terminal

Sometimes a command runs without a full environment. Programs that run as part of a CGI script or programs run by schedulers like `cron` don't actually have a terminal to run in. Your average Unix-like system doesn't fire up a shell session to run these commands, but instead runs them as child processes. If you don't want automated processes running arbitrary commands via sudo, look at `requiretty`.

The `requiretty` option tells sudo to only work if the command is run in a terminal. Enabling this option in sudoers means that programs cannot run without a terminal. A user can't write a CGI script that calls sudo – well, okay, they can *write* it, but the sudo call won't work.

You can now manage the environment sudo creates, or whether it needs an environment at all. Now let's see how sudo can protect your users from a damaged system.

Chapter 9: Sudo for Intrusion Detection

One of the problems mentioned in the previous chapter is that of tampering with the user's path. Sanitizing the path helps, but then our intruder might replace the actual */usr/bin/passwd* command with his own treacherous version. Sudo 1.8.7 and later can verify the cryptographic *digest* (or *checksum*, or *hash*) of a command before running it, preventing these kinds of attacks.

Why is this useful? A cryptographic digest is a mathematical transformation that creates a fixed-length string for any piece of data, such as a file. Even minor changes in the source file dramatically change the generated digest. If sudo knows that the correct cryptographic digest for the legitimate passwd command is X, but the passwd command on the disk has a digest of Y, sudo will refuse to run the command. For more about cryptographic digests, check out my book *PGP & GPG* (No Starch Press, 2006).

An intruder is not the only one who might alter the file

containing a command. If you have write access to the directory containing the command, you might accidentally alter it yourself. Similarly, digests can protect you from users who chafe at their restrictions. "I know how to fix this, I just need root!"[7]

Digest verification can prevent you from running a copy of `dd` that someone accidently copied over the `mv` command. Would running that hurt anything? Probably not, unless you intended to move some very oddly named files. But such errors can be catastrophic, and they're the first sign that this operating system instance is badly damaged. You want as much early warning as possible of system damage.

Using digests for command integrity verification has two components: generating the digest, and writing a sudoers rule that validates the digest.

Generating Digests

Different Unix-like operating systems have different commands for computing cryptographic digests. (Because they can.) Rather than suggesting the `sha512` command only for you to discover that you need `sha512sum` because you're using Linux, I recommend the more generic `openssl` tool for generating checksums.

Sudo supports several different variants of the SHA digest algorithm: SHA-224, SHA-256, SHA-384, and SHA-512. Higher numbers mean that the digest is more difficult to reverse-engineer, but creating and validating the digest also takes more computation power. Also, digests get much longer as the strength increases. SHA-224 provides sufficient protection against all realistic attacks with today's hardware.[8]

```
$ openssl dgst -sha224 /usr/bin/passwd
SHA224(/usr/bin/passwd)= c6eab09e527dc…
```

The 56-character string after the equal sign is the SHA-224 digest of the file */usr/bin/passwd*. Most programs will have unique

digests. Some programs have multiple names – for example, the sendmail command is also known as newaliases, mailq, hoststat, purgestat, and probably a few other names. (I have my own preferred names for sendmail, but children might stumble across this book.) You can list all of those names in a sudoers alias. Which takes us to the next topic.

Digests in Sudoers

Use a cryptographic digest much like other tags. After the equal sign put the type of digest, a colon, and the digest itself, then the command list. Unless you have multiple commands with identical digests, you probably need one rule per permitted command. As SHA-224 digests are 56 characters long, I've truncated the actual digest in all of these examples.

```
mike ALL = sha224:d14a028c… /usr/bin/passwd
```

When I ask sudo to run `passwd`, sudo computes the SHA-224 digest for */usr/bin/passwd*. If the generated digest matches the digest in the sudoers rule, sudo will run the command. Otherwise, you'll get the generic "not allowed" message. If `sudo –l` shows that you have permission to run a command, but every attempt to run the command gets the "not allowed" message, the checksum on the command doesn't match the file's checksum in sudoers.

If multiple binaries have the same digest, you probably made a mistake somewhere. Double-check your `openssl` command. If

multiple program files really do have the same digest, they might be the same program in disguise – e.g., sendmail and its posse. You can list commands with identical digests together like so:

```
Cmnd_Alias SENDMAIL = sha224:65f81… /usr/sbin/sendmail, \
    /usr/bin/mailq, /usr/sbin/hoststat, /usr/bin/newaliases
```

If you want to compute the cryptographic digest of every legitimate binary on your system, I recommend writing a script to do so. If the script lets you predefine groups of commands for command aliases, so much the better.

Digests and Multiple Operating Systems

Once you centralize your sudoers policy, you might find that you need a policy that permits multiple digests for a single command. The sendmail commands on Ubuntu will have different digests than the sendmail commands on FreeBSD, and those on FreeBSD 9.2 will differ from FreeBSD 9.3.

How can you cope with this? Use one command alias per operating system.

```
Cmnd_Alias FB92_SENDMAIL = sha224:65f81… /usr/sbin/sendmail, \
    /usr/bin/mailq, /usr/sbin/hoststat, /usr/bin/newaliases
Cmnd_Alias PRECISE_SENDMAIL = sha224:213ff… /usr/sbin/sendmail, \
    /usr/bin/mailq, /usr/sbin/hoststat, /usr/bin/newaliases
Cmnd_Alias SENDMAIL = FB92_SENDMAIL, PRECISE_SENDMAIL
```

Did I mention using a script to generate digests for your operating system?

You won't want to recompute this on every machine on your network. It's much better to design this policy once and distribute it to the rest of the network, as we discuss in the next chapter.

Chapter 10: Sudoers Distribution and Complex Policies

Sudo is a lot of trouble for a single machine. If you run hundreds or thousands of systems, however, sudo makes user privileges manageable. Not easy or simple, but manageable. The best way to have a consistent policy across your network is to write a single sudoers file and replicate it to all machines on the network. While it's fairly simple to do this, here are a few hints on writing and deploying safe and secure policies.

Breaking Sudo

We've touched on how to escape sudo's restrictions earlier in this book, but let's consider them all together. The following is a "greatest hits" of how to write sudoers policies.

Do not exclude commands from an alias. Users can easily bypass command lists like `ALL, !/bin/sh`. Using the ALL command list gives people privileged access, no matter how the system owner tries to restrict it.

Use the NOEXEC flag by default in your command lists. Specifically enumerate commands that must run other commands. You'll have a rough few days as users call to complain that they can't run certain commands, but you'll quickly find the commands that legitimately must run other commands. When you automatically distributing a single sudoers file across the network, those changes will quickly propagate to all hosts.

Use aliases for users, commands, hosts, and RunAs settings. Use the alias rather than the command name in your rules. This

simplifies changes and helps ensure all your users have identical access to others in their group.

Most ways to escape restrictions can be eliminated with proper configuration. "Proper configuration" usually means "spell out exactly the permitted access." Don't just give people unlimited access to all commands; instead, sort out who should be doing what and what access they need to do their real jobs. Yes, this means spending time and energy having face-to-face conversations with living human beings who have their own opinions and desires, rather than doing the fun computing stuff.

Hesitate to give root-level privileges to shell scripts via sudo. While sudo sanitizes the user's shell environment, a shell script can put that scary stuff right back in. In too many cases, running a shell script as root via sudo is equivalent to giving the user root. Even if you use cryptographic digest verification to ensure that the script runs unedited, shell scripts often pull often in other shell scripts. Users and intruders can subvert any number of shell scripts with

environment variables. Don't think your users are different and won't mess around with your carefully written shell scripts. They aren't and they will.

On some hosts, a tight sudo configuration isn't realistic. Desktop machines run lots of programs that run other programs. A user who has physical access to the machine and needs to run a graphic desktop can get root-level access on the machine without much difficulty. Your best practice is to assume that desktop machines are not trustworthy, and secure your servers against rogue workstations as well as external intruders.

If you're not willing to do the work of creating a real sudoers policy, then don't waste your time slapping together a half-cooked sudoers policy that sort, of more or less, kind of, does what you want, basically. Instead, give users unlimited access and deal with the fallout. After enough unnecessary downtime, system damage, and lost nights and weekends, you'll develop a willingness to write a real sudoers policy. Logging user activity (see Chapter 12) can

help assess exactly what happened when things go wrong, and might be a good replacement for your organization.

Hostnames and Sudoers

When managing sudoers individually on each machine, the hostname part of the policy tends to disappear from the sysadmin's view. It's still in the file, but your conscious mind no longer sees it. It's just "that 'ALL =' *thing*" that must appear in the middle of every rule. I haven't given it much attention so far, because we've only considered single-system policies. When you want to use a single sudoers file across your entire network, suddenly the hostname field becomes much more important.

Sudo gets the name of the local machine by running hostname. The hostname in your sudoers policy must exactly match whatever hostname the local machine thinks it is. This can cause difficulty in heterogenous networks. My Linux servers usually have a hostname consisting of a single word, such as **www8** or **sip2**. My BSD machines have a hostname that includes the domain, such as **www.michaelwlucas.com.** Before you start writing a centralized sudoers policy, investigate your naming scheme as it is actually deployed

on the real servers. Are they consistent? If you're using centralized server provisioning, you're probably okay. If you're still running artisan-managed servers, or you install servers by hand, you have inconsistencies.[9] Address those inconsistencies before you build your policy. Or use DNS or IP addresses.

DNS and Sudo

The Domain Name System maps hostnames to IP addresses. A server might think its name is **www8**, but the DNS records it as **www8.michaelwlucas.com**. DNS is centrally managed (mostly; more on that later). Having sudo refer to DNS for machine names removes any local host name inconsistency issues. It also adds a dependency on DNS for machine management. If your DNS servers fail, sudo will not work. If sudo won't run because DNS is down, and you can't restart DNS because sudo is down, congratulations! You failed to think through your failure modes. Expect your local Thea to come for your carcass shortly.

Hosts might be configured to resolve IP addresses and hostnames from a variety of information sources, such as YP or LDAP. If the server prefers one of these information sources to DNS, then you need to verify that your sudoers rules match the hostname in that information source. The most common alternate information source is the hosts file, */etc/hosts*. Check to see if your server prefers the hosts table to DNS, and confirm the server's name in that file if so.

A machine can have multiple host names in both DNS and hosts, but sudo only uses the primary host name. Sudo ignores all aliases or additional records. If you're using the hosts file, only the first host name in an entry is used. If you're using DNS, any CNAME records are ignored. Sudo only uses the hostname as shown in forward and reverse DNS.

To enable the use of DNS, use the fqdn option in sudoers.

```
Defaults fqdn
```

Sudo still checks the local host name, and if the sudoers rule

happens to match the local name, the rule matches. If the name doesn't match, sudo uses DNS and compares each rule to the server's fully qualified domain name. Rather than using the short hostname **www8**, you'll need the full hostname.

```
%helpdesk www8.michaelwlucas.com \
    /usr/bin/passwd [A-z]*, !/usr/bin/passwd root
```

The lines in your sudoers file will be much longer, but that's okay. Also, your sudo commands will take a little longer as sudo queries the DNS for the local host name.

The obvious way to break hostname-based protections, however, is for the system administrator to change the local host's name. If your sudoers policy permits an otherwise unprivileged user to change the machine name, then he can change the policy applied to the machine.

IP Addresses

I find that using IP addresses in my sudoers policies is more reliable than using hostnames, at least in my environment. On a

large network, where machines exist on different segments and have different network access rules, system administrators usually have no access to the network equipment. A rogue sysadmin might change the name of a web server to that of a host on the database tier, but he cannot change the IP address of that server without losing access to the machine.

Use host aliases to define these network subnets.

```
Host_Alias WEBSERVERS 192.0.2.0/24
Host_Alias DBSERVERS 203.0.113.0/24
```

Assign access rules to these host aliases, and the only way a problematic user can get around the access controls is to move the machine to another subnet. Ultimately, how you design your sudoers policy to avoid these hostname changes depends on your staff and users, your environment, and your risk tolerance.

Including Files in Sudoers

A sudoers policy can include other files by reference. This lets you have a generalized sudoers policy for all your systems, and add other files by machine role or functions. You can add specific files, files by hostname, or files within a directory by using an #include statement.

The file is inserted into the sudoers policy at the spot that you use the include statement. If you include files at the top of sudoers, your global rules override anything in the included policy. If your include statement appears last in sudoers, then the included file overrides the global policy. Why is this important? Think about an included file with this line:

```
%wheel ALL = !ALL
```

The **wheel** group is traditionally those users permitted to use the root password – also known as "the senior sysadmins." Depending on your operating system, this might be the **admin** group or something else. The included file forbids all users in **wheel** to run

any commands via sudo. If this rule appears last in the sudoers policy, it removes the senior sysadmin's access to the servers. This is probably not what you want.

Include Specific Files

Maybe you have a base template of a sudoers security policy that you distribute to all systems, so that your senior systems administrators can access all servers. Individual machines have their own security policies tailored to the system's needs. In this case, you would copy */etc/sudoers* to all machines on the network, and tell local users to put their own rules in a different file, such as */etc/sudoers.local*. Add an #include statement to your global sudoers.

```
#include /etc/sudoers.local
```

Set your local additions in that file.

Per-Host Include Files

Maybe you want to include a file based on the local host name. You can use the %h escape character to use the local host name in a

file.

```
#include /etc/sudoers.%h
```

On the machine **www8**, sudo would look for a file called */etc/sudoers.www8*.

Including Directories

Including one file isn't enough for you? Sudo lets you include all the files in a directory by using the `#includedir` statement.

```
#includedir /etc/sudoers.d
```

Many Linux distributions use this type of syntax. The idea is that you can have a central, standard sudoers policy, and then copy additional policies to a machine based on the machine's function. The host is a webserver? Copy your standard file *001-sudoers.www* to the include directory. Database server? Copy the database file. Both? Then copy both.

This is a perfectly valid way to manage a sudoers policy. By the time your network grows this complex, however, you're much better off investigating an LDAP-based security policy (Chapter

11) instead of managing sudo by local files.

Sudo reads and processes these files in lexical order. In lexical order numbers always sort before upper case letters, and upper case letters always sort before lower case letters. Lower case letters come before accented characters. You've seen this kind of ordering every time you run a plain `ls` in a directory. You'll see numbers sort like `1`, `11`, `12`, `2`, and then `21`. The word `Rat` comes before `gerbil`. The easiest way to control sorting is to have all of your included files start with numbers, and include the leading zeroes. That way, policy file *001-sudoers.www* will get processed before *100-sudoers.database*. File *2-sudoers.wordpress* gets processed after both, so include those leading zeroes.

Or use an LDAP-based policy to show a single consistent policy to each machine. You'll be happier… eventually.

Errors in Include Files

If a file included in */etc/sudoers* is syntactically invalid, sudo will not

run – precisely as if you had a syntax error in */etc/sudoers* itself. Visudo only checks the integrity of one file, not everything included in the sudoers file. Use the –f flag to aim visudo at a different file.

```
# visudo –f /etc/sudoers.www8
```

Visudo will open a copy of this file, edit the copy, check the file's syntax, and either replace the original file or tell you to fix your errors, exactly as it does for */etc/sudoers*.

Single Sudoers Across the Network

If you run hundreds of machines, you already have a way to distribute files to all of them. Tools such as Puppet, Chef, Ansible, or even rdist, make this almost easy. Configuring sudo on a central machine and pushing the sudoers file out to all of the hosts in the network does not prevent someone from editing a local machine's sudoers file. But it improves detection of and recovery from such changes. It's also easier than using an include directory – you can put your various servers in groups and use those groups for rules.

If you're centrally managing sudo, I strongly recommend having each local machine validate that it can parse the new sudoers file before installing it as */etc/sudoers*. If you install a sudoers file that works on sudo 1.8.9 on a machine running sudo 1.8.7, you might have included options or rules that the older sudo cannot parse. If sudo cannot parse */etc/sudoers*, sudo will not run. Validating the new file with visudo –cf before copying it into place will save you a lot of trouble. I strongly recommend reading Jan-Piet Mens' blog post

"Don't try this at home: /etc/sudoers" (http://jpmens.net/2013/02/06/don-t-try-this-at-the-office-etc-sudoers/) and the related posts for a very good description of exactly how much pain a bad sudoers policy causes on a large network. (It's amusing because it happened to someone else.) Mens also has an Ansible playbook for safely distributing */etc/sudoers* so you can learn from his suffering.

While configuring your sudo policy in one location and pushing it to all your hosts has distinct advantages over configuring it separately on each host, better still is having sudo read its policy from LDAP.

Chapter 11: Security Policies in LDAP

One problem with sudo is that it's normally configured on the local machine. An intruder (or a clever but very naughty user) who leverages his way into altering the sudoers file can alter his own permissions. This is bad. The way to eliminate this risk is to remove the sudoers policy from the machine.

The Lightweight Directory Access Protocol (LDAP) provides common information across a network. While it usually stores usernames and passwords, it can support any arbitrary directory-style information. A sudo security policy fits well into LDAP.

The advantage of having your sudoers policy in LDAP is that a user who compromises a machine cannot alter the sudoers policy. Even gaining root on a server doesn't give him access to a read-only LDAP server. Also, changes to an LDAP-based security policy immediately propagate to all the machines on the network.

Typos cannot prevent sudo from running, as they can with

sudoers. An LDAP server will not accept improperly formatted data. You can mistype machine and user names, but any sudo configuration you stuff into your LDAP server is syntactically valid.

The disadvantages of configuring sudo from LDAP? First, you must have an LDAP server. When that LDAP server fails, your authentication and sudo security both die with it, so you probably want more than one. You must have a sudo install that supports LDAP, which isn't usually in the default install but is easily obtained.

Sudo includes very detailed documentation on using LDAP as a security policy provider in the documents *README.ldap* and the *sudoers.ldap* manual page. Read those documents before planning your deployment. This book does not replace the official sudo documentation, but provides context, guidance, and an overview parallel to that documentation. I don't cover details like AIX using */etc/netsvc.conf* instead of */etc/nsswitch.conf*; for that, you need your

operating system manual or the official sudo documentation.

Sudoers Policies versus LDAP Policies

Building a sudo security policy for LDAP is different than creating an sudoers-based policy. First off, LDAP sudo policies do not support aliases. The user aliases, command aliases, and soforth that we spent a chapter on earlier in this book? Not applicable to LDAP-based policies. Instead, use LDAP groups for users and servers. This isn't necessarily an advantage or a disadvantage, but you need to know about it. The design of LDAP means it's very easy to add a new command, user, or host to a rule, however.

A sudoers-based policy works on a "last match" basis, so you can put generic rules at the top of the policy and get more specific further on. LDAP doesn't automatically deliver query results in a deterministic order. You can order your individual sudo rules in LDAP, placing one rule before another so that "last match" works, but it's an extra step to remember. You cannot order attributes within a single LDAP sudo rule.

Finally, LDAP-based policies don't use negations for hosts,

users, or RunAs. Negations on commands work exactly as well as negations do in sudoers – poorly. Remember that you cannot order attributes with a single sudo rule, so if there's a conflict, any command negation takes precedence. Save yourself the indigestion. Don't use negations with LDAP sudo policies.

Prerequisites

This is not a book on LDAP. If you don't know what a schema or an LDIF is, this section will baffle, annoy, and possibly scare you. That's because LDAP can baffle, annoy, and scare the uninitiated. Skip ahead to Chapter 12. Logging sudo activity is much more interesting and useful than it sounds, and you don't need any external infrastructure to do it. This chapter focuses on LDAP-based sudoers policies and attaching the sudo client to LDAP.

Site requirements vary too much for me to take you through a "generic" LDAP configuration. As OpenLDAP is the server most commonly used for sudo, I'll use it for specific detailed examples, but I'll touch on other supported LDAP servers.

I assume that you have LDAP-based authentication working, that your setup is secure and stable, and you have both the ability to import LDIF files and to make minor changes through an LDAP browser. I assume that you're using the same LDAP servers for sudo as for authentication. The sudoers policy in LDAP should not

be writable by the sudo clients it serves – otherwise, one compromised machine can rewrite the sudoers policy for all the systems on the network. Similarly, I don't let my LDAP client servers have any write access to the LDAP server, requiring users to go to a specific host or interface to change their passwords and other account information.

I also assume that you have a sudoers-based policy to start with. It doesn't need to be a big policy – even something simple like "here are some defaults, and this group gets full access" will get you rolling.

If you don't have LDAP-based authentication, stop trying to stuff sudo into LDAP. You've gotten ahead of yourself. Get your machines pulling their user and group information from and authenticating against LDAP. Then return here and try again.

We'll start with your sudo client, and then proceed to the LDAP server.

LDAP-Aware Sudo

An LDAP-aware sudo works without a sudo policy in LDAP, so installing the LDAP-aware sudo is the sensible place to start. Most operating systems have a package for sudo built with LDAP support or allow you to easily enable it. Debian-based systems have a sudo-ldap package. CentOS-style systems allow you to enable LDAP for sudo in */etc/nsswitch.conf*. On FreeBSD you must build your own sudo package to enable LDAP, but the ports system makes that pretty easy. Check your operating system documentation, and follow the instructions to get an LDAP-capable sudo installed on your system.

Then configure the LDAP server to serve and recognize sudo data.

Add Sudo Schema to LDAP server

An LDAP server that supports sudo policies must understand the syntax and structure of those policies. A *schema* defines a data structure for an LDAP server. Each vendor's LDAP server product has its own schema system that is (of course) subtly incompatible with all the other LDAP servers. Sudo includes three LDAP schemas for three LDAP servers in the files *schema.OpenLDAP* (for OpenLDAP servers), *schema.ActiveDirectory* (for Microsoft servers), and *schema.iPlanet* (for Netscape-derived servers). Some operating system packagers include the sudo schema in their LDAP server, so check for it before trying to install your own.

After you add the schema to any of these LDAP servers, index the sudoUser attribute. This greatly accelerates sudo lookups.

Next, I'll briefly touch on adding the schemas to all three LDAP servers.

Adding Sudo to OpenLDAP

To add the sudo schema to OpenLDAP, copy the schema to your schema directory (usually */etc/openldap/schema/*) as the file *sudo.schema*. Then add the following lines to *slapd.conf*. You probably want to place these statements near the other schema and index statements:

```
include /etc/openldap/schema/sudo.schema
index sudoUser eq
```

Restart slapd, and OpenLDAP will support sudo policies.

Adding Sudo to iPlanet

Copy the schema file to the server schema directory. This directory varies by operating system, so check your server documentation. Give it the name *99sudo.ldif*. Restart the server.

Now use your LDAP browser to create a Service Search Descriptor for sudoers.

```
serviceSearchDescriptor: sudoers: ou=sudoers,dc=example,dc=com
```

You're ready.

Adding Sudo to Active Directory

Copy the Active Directory schema file to a domain controller, and run the following command.

```
C:> ldifde -i -f schema.ActiveDirectory -c dc=X
dc=example,dc=com
```

That's it.

Creating Sudo Policy in LDAP

The sudo policy needs a container and an initial policy. Here's how to handle each.

Sudoers Container

Your sudo policy needs an LDAP container. Most LDAP administrators have very definite ideas about where new containers for add-on software belong. Obey her wishes in the matter – LDAP causes her enough grief, she doesn't need any lip from you. For reference, here are the default locations for each major server:

OpenLDAP: ou=SUDOers, dc=example, dc=com

Active Directory: cn=sudoers, cn=Configuration, dc=example, dc=com

iPlanet: ou=sudoers, dc=example, dc=com

Despite calling the container "sudoers," remember that an LDAP-based policy doesn't work quite like a sudoers file.

Here's an LDIF for a sudo container for the OpenLDAP server

for **mwlucas.org**. For other servers or other container locations, change the Distinguished Name path.

```
dn: ou=SUDOers,dc=mwlucas,dc=org
objectClass: top
objectClass: organizationalUnit
ou: SUDOers
```

Import this into your server, either through the command line or through your browser. Now you can create your initial LDAP sudoers policy.

Converting /etc/sudoers to LDAP

The convenient thing about switching from an */etc/sudoers* policy to an LDAP-based policy is that you don't need to create the LDAP entries from scratch. You can convert an existing sudoers file to an LDAP-friendly LDIF file with the script sudoers2ldif, included in the sudo suite. It's a Perl script, usually installed as part of an LDAP-aware sudo package.

Before running sudoers2ldif, you need to set the SUDOERS_BASE

environment variable to the location of the sudo policy container. The command uses this variable to put the created LDIF in the correct part of the directory tree.

```
$ SUDOERS_BASE=ou=SUDOers,dc=mwlucas,dc=org
$ export SUDOERS_BASE
```

Now run sudoers2ldif, giving your sudoers file as an argument.

```
$ sudoers2ldif /etc/sudoers > /tmp/sudoers.ldif
```

This spits out an LDIF version of your sudoers policy. One nice feature of sudoers2ldif is that it fills in the sudoOrder attribute, ordering your rules so that the "last match" rules processing works. See "Sudoers Policies versus LDAP Policies" earlier in this chapter for details.

You could just import this file into your LDAP server and be done with it, but that would leave you blindly trusting that the script worked. Let's see what kind of entries your sudoers file becomes.

Sudoers into LDIF

Let's start with a very simple */etc/sudoers*.

```
Defaults env_keep += "HOME SSH_CLIENT SSH_CONNECTION \
 SSH_TTY SSH_AUTH_SOCK"
%wheel,%sysadmins ALL=(ALL) ALL
```

We retain several environment variables to allow SSH agent forwarding, and then we allow anyone in the groups **wheel** and **sysadmins** to run all commands via sudo. Essentially, this sudo policy replaces `su` with `sudo`.

What does this become as an LDIF? We will go through descriptions of all the various schema fields later, but the generated LDIF is pretty easy to understand. We'll look at each entry separately.

```
dn: cn=defaults,ou=SUDOERS,dc=mwlucas,dc=org
objectClass: top
objectClass: sudoRole
cn: defaults
description: Default sudoOption's go here
sudoOption: env_keep += "HOME SSH_CLIENT
    SSH_CONNECTION SSH_TTY SSH_AUTH_SOCK"
sudoOrder: 1
```

This entry is named "defaults," according to the dn statement. The objectClass statements attach this to the sudo policy. The sudoOption statement gives the actual sudo rules. Finally, sudoOrder puts this sudo rule first in the list of rules to process.

Here's the sudoers line giving two groups permission to run all commands as **root,** written as an LDIF.

```
dn: cn=%wheel,ou=SUDOERS,dc=mwlucas,dc=org
objectClass: top
objectClass: sudoRole
cn: %wheel
sudoUser: %wheel
sudoUser: %sysadmins
sudoHost: ALL
sudoRunAsUser: ALL
sudoCommand: ALL
sudoOrder: 2
```

This rule has two sudoUser entries, one for each group the rule applies to. There's a sudoHost entry to show this rule applies to all hosts, and a sudoRunAsUser indicating that this rule lets these users run commands as all other users. The sudoCommand entry

lists all the commands this rule covers.

Remember that entries appear within an item in no particular order. This rule has two sudoUser entries, one for **wheel** and one for **sysadmins**. The **wheel** group happens to appear first in this list, but in a live query it might be reversed. If order is important, you need to make a second rule and put it in order using the sudoOrder attribute.

You can import this initial policy into your LDAP server, then configure the sudo client to pull information from LDAP.

Activating Sudo Client LDAP

Your LDAP-aware sudo client has the ability to ask LDAP for security policies, but it probably won't do that by default. You must tell sudo where to find the LDAP-based policy, and then configure sudo to use that policy.

Finding the LDAP Policy

I said earlier that I assume you have a working LDAP setup. This means that your local machine can pull user and group information and passwords from your LDAP directory. This simplifies sudo configuration, because you only need to worry about the sudo portion of LDAP.

Start by running sudo –V to ask your sudo install where it expects to find its LDAP configuration file.

```
$ sudo -V | grep ldap
…
ldap.conf path: /etc/ldap.conf
ldap.secret path: /etc/ldap.secret
```

This particular sudo install expects to find *ldap.conf* and *ldap.secret* in */etc*, the default for this operating system.

Most operating systems can share a single *ldap.conf* between all applications. This lets your sudo install piggyback on your working LDAP configuration. Some operating systems use sudo-specific LDAP configurations. For these operating systems, you can usually copy the basic LDAP configuration from the main system file to the sudo-specific file. Check your operating system manual if you have any concerns. [10]

Now add the sudo LDAP configuration to your sudo's *ldap.conf*. Sudo accepts three different *ldap.conf* statements, but only sudoers_base is mandatory.

sudoers_base: This is the mandatory location of the sudoers policy. You can have multiple sudoers_base entries. Sudo will query them in the order given in *ldap.conf*.

sudoers_search_filter: This is an optional LDAP search filter to reduce the number of results returned by an LDAP query. Sudo

works fine without this filter.

sudoers_timed: This is a yes (or true, or on) or no (or false, or off) setting to tell sudo to check to see if a sudoers rule has expired. See "LDAP Policy Expiration" later in this chapter.

The standard *ldap.conf* entry for sudo on my network looks like this:

```
sudoers_base ou=sudoers,dc=mwlucas,dc=org
```

Old documentation mentions setting sudoers_debug in *ldap.conf*. This is deprecated, and the setting will be buried in an unmarked grave before long, so don't start using it now. To log sudo's interactions with LDAP, use the LDAP logging subsystem described in Chapter 12.

Now that your LDAP clients can find the sudo policy, tell sudo to look at LDAP.

Sudo and nsswitch.conf

Use */etc/nsswitch.conf* to tell sudo to look at LDAP. The name service switch configuration file usually tells programs where to look for

information such as hostnames and usernames. Sudo gets lumped in with the rest of them. Use an entry like this to tell sudo to check LDAP:

```
sudoers: ldap files
```

Sudo will check the information sources in the order listed here – first LDAP, then */etc/sudoers*. If your sudo install should never use the local sudoers file, remove the files statement from this line. You should also add the ignore_local_sudoers option to your LDAP policy, as we'll see later.

Sudo Rules and Roles

A one-line sudo policy in /etc/sudoers becomes a single LDAP entry, called a *sudoRole*. Both of the entries we looked at in the "Sudoers into LDIF" section earlier are sudoRoles.

All sudo attributes have specific permitted values, most commonly usernames, groups, or commands. You cannot enter an invalid data type into an attribute – an attribute that expects a username won't accept an IP address, and the LDAP server will reject it if you try to set it incorrectly. Mind you, the LDAP server can't know if **mike** is a hostname or username, so you must verify that the syntactically-valid rule you just wrote is the rule you want to write. The one special value is ALL, which matches all possible entries for that attribute.

All sudoRoles have the Distinguished Names (DN) attribute, the sudoRole objectClass attribute, and the Common Name (CN) attribute. LDAP needs them, after all. But three additional attributes must appear in every sudoRole, and a few optional

attributes can appear when useful. The three mandatory attributes are sudoUser, sudoHost, and sudoCommand.

sudoUser

The sudoUser attribute is a user name, exactly like those used in a sudoers policy. Remember, you cannot use aliases in a sudoUser attribute. You can use operating system groups, group IDs, and netgroups. If you want to use non-system groups in LDAP rules, you must add a plugin for them to each local sudo install. Groups stored in LDAP work fine, so use them rather than jumping through these extra hoops. Each user name must appear in its own sudoUser entry within a sudoRole.

```
sudoUser: %wheel
sudoUser: mike
sudoUser: kurt
```

sudoHost

This is a list of hosts, with the same syntax and restrictions as a

host entry in a sudoers rule. You can use host names, IP addresses and networks, and netgroups. ALL matches all hosts.

```
sudoHost: 192.0.2.0/24
sudoHost: www.michaelwlucas.com
sudoHost: +dbservers
```

sudoCommand

This is the full path to a command, plus any command-line arguments and wild cards. This is exactly like the command list in sudoers, except that you cannot use aliases. ALL, just as in sudoers, matches all commands.

You can use the word `sudoedit` followed by a file name or path to permit use of sudoedit on those files. Similarly, putting a digest algorithm and a digest before a command tells sudo to verify the digest before running the command.

```
sudoCommand: sudoedit /etc/namedb/named.conf
sudoCommand: sha224:d14a028c… /usr/bin/passwd
sudoCommand: /sbin/dump
sudoCommand: /sbin/restore
```

In addition to the sudoRole's three mandatory attributes, LDAP-based policies have four optional attributes that let them fully emulate sudoers policies: sudoRunAsUser, sudoRunAsGroup, sudoOptions, and sudoOrder.

sudoRunAsUser

The sudoRunAsUser attribute gives a list of target users that sudo users can run commands as. This works exactly like the RunAs list (see Chapter 4) for sudoers. The word ALL matches all users. sudoRunAsUser also accepts user ID numbers, groups, or netgroups. List each target in its own sudoRunAsUser entry.

```
sudoRunAsUser: oracle
sudoRunAsUser: postgres
```

sudoRunAsGroup

This attribute permits users to run commands as a member of a group. The groups have the same valid names as groups within a sudoers policy. List each target group on its own line.

sudoRunAsUser: operator

sudoOrder

This attribute assigns role number. Roles are processed in order, from lowest to highest. SudoOrder lets you emulate the last matching rule behavior from a sudoers policy. A sudoRole without a sudoOrder has a sudoOrder of 0, and so is processed first. If you have multiple sudoRoles without a sudoOrder, they are processed in the order served up by LDAP – that is, randomly.

sudoRole Times

LDAP-based policies let you set activation and expiration dates and times for a sudoRole, a feature you won't find in sudoers-based policies. Sudo checks for activation and expiration timestamps only if you have the sudoers_timed option in *ldap.conf*. Without this option, sudo ignores times.

The sudoRole attributes sudoNotBefore and sudoNotAfter control sudoRole timing. These attributes accept a value of a four-digit year, followed by two digits each for month, day, hour, minute, second, and a one-digit tenth of a second. Or, if you prefer, YYYYMMDDHHMMSSZ. The date and time are in Coordinated Universal Time (UTC), not your local time zone.

```
sudoNotBefore: 201401011300000
sudoNotAfter: 201401312200000
```

The sudoRole for the example above becomes valid on 1 January 2014 at 13:00, and expires on 31 January 2014 at 22:00. These times look weird, but my site is five hours ahead of UTC.

The rule becomes valid at 8 AM local time, and expires at 5PM on the last day.

This sudoRole is not valid until the date and time in the sudoNotBefore attribute. It is no longer valid after the sudoNotAfter attribute.

If you have multiple sudoNotBefore and sudoNotAfter attributes, the most permissive entry is used – that is, the earliest sudoNotBefore and the latest sudoNotAfter. If you try to put in two separate time ranges, the sudoRole will permit access from the earliest start time to the latest end time. If you put in a sudoRole that says "This rule is valid for the first 10 days of September" and another sudoRole that says "This rule is valid for the last ten days of October," the user will get access from the first of September to 31 October. Remove obsolete sudoNotBefore and sudoNotAfter attributes from your directory.

A role with useless dates never gets used.

objectClass: sudoRole
cn: mwlucas

sudoUser: mwlucas
sudoHost: ALL
sudoCommand: ALL
sudoNotBefore: 201402030000000
sudoNotAfter: 201402301200000

Here, Thea has granted me total access to all systems for twelve hours. On the 30th of February.

Disabling sudoers

The point of putting security policies in LDAP is so that users who finagle their way into editing */etc/sudoers* cannot write rules that give themselves more access. We configured sudo to look at LDAP first for its policy, which is a good step. Now we need to decide if we want to have a local sudoers file.

If we have a local sudoers policy file, users might figure out how to edit it. If LDAP tells sudo to ignore the local sudoers policy, it doesn't matter if users edit sudoers or not; they don't get extra access. The risk you get is that when your LDAP systems fail, you'll lose sudo access on your LDAP clients. See "LDAP Caching" later this chapter for possible solutions.

Tell sudo to completely ignore */etc/sudoers*. with the ignore_local_sudoers option in LDAP. Add ignore_local_sudoers to your default policy. The standard location for this policy on the OpenLDAP server for a domain would be at the Distinguished Name cn=defaults,ou=sudoers, dc=example,dc=org

```
dn: cn=defaults,ou=SUDOERS,dc=example,dc=org
objectClass: top
objectClass: sudoRole
cn: defaults
description: Default sudoOption's go here
sudoOption: env_keep += "HOME SSH_CLIENT
    SSH_CONNECTION SSH_TTY SSH_AUTH_SOCK",
    ignore_local_sudoers
sudoOrder: 1
```

When sudo sees this option in LDAP, it stops looking at the local sudoers file.

Do you want to disable local sudoers policies? Probably. An LDAP client without LDAP won't function properly anyway, so you'll many more problems. The option is yours, however.

Learning sudoRole policies

If managing LDAP isn't your main job, but you want to support sudo policies via LDAP, you get to learn a new skill. Once you understand writing sudoers security policies, expressing the same thing in LDAP isn't that much harder.

If you get confused, sudoers2ldif is your new friend. You want to know how to write an LDIF version of a particular sudoers rule? Write a one-line sudoers file that contains only your desired rule, then run sudoers2ldif to see the result. Modifying an example LDIF is much easier than writing one from scratch. Soon, you'll be writing and editing sudoRole LDIFs effortlessly. Don't tell the LDAP administrator you can write LDIFs, however, or she might try to suck you into writing more of them for other people.

I know people who use LDAP to distribute their sudo policies but actually write the policies in sudoers format and then use sudoers2ldif to generate the LDAP configuration. This automatically handles rule ordering with sudoOrder. This is a

perfectly acceptable solution, and if you're not comfortable with LDAP it might even be advisable.

LDAP Caching

The big risk when using LDAP for authentication and policy distribution is that your network becomes dependent on the LDAP servers. Hopefully you have at least two LDAP servers, distributed in such a way that they resist most failure scenarios. And hopefully you have enough LDAP servers that a failure of a substantial fraction of them won't overload the survivors.

You can choose to cache LDAP information locally on each machine, to tide the servers through a brief outage. The System Security Services Daemon (SSSD) provides caching services. SSSD is a fairly new program created as part of the Fedora project, and its support for non-Linux systems is mixed but improving.

As of sudo 1.8.4, you can build sudo with SSSD support. Sudo-sssd lets you add SSSD as an additional information source via */etc/nsswitch.conf*. This lets sudo reference the cached security policy even if the LDAP servers are down. You can configure SSSD to proactively download the sudo policy from the LDAP server so it's

prepared for an LDAP failure.

Most operating systems don't have a package for sudo with SSSD support. If you're using SSSD, consider using it for sudo as well. Given SSSD's mixed support on every operating system except Linux, I'm not going to cover it in detail here. If SSSD supports your operating system, you can find a useful tutorial on using sudo with SSSD at http://jhrozek.livejournal.com/2065.html.

Now let's look at sudo logging. It's more useful than you think.

Chapter 12: Sudo Logging & Debugging

You can now control what access people have to privileged commands. Everything's good, right? Certainly... until the day you walk in and find half your servers hanging because their */usr* filesystems have fled for parts unknown. Everybody will want to know who the idiot was. Sudo has three different logging mechanisms: a simple "what sudo did" log via syslogd, a debugging log, and a full session capture log. Sudo can also notify the system owner when users succeed or fail to run commands.

Sudo and Syslogd

Sudo logs user activity through the standard syslog protocol. On your average Unix-like system, sudo logs show up in a file like */var/log/messages* or */var/log/syslog*. Here's a typical sudo log message:

Aug 27 23:34:44 www9 sudo: mike: TTY=pts/1 ; PWD=/home/mike ; USER=root ; COMMAND=/usr/bin/passwd carl

We have the date and time someone ran sudo, and the machine name (**www9**). Then we have the user who ran sudo (**mike**), the terminal he was on (**pts/1**), the directory he was in (*/home/mike*), who he ran the command as (**root**), and the command he ran (/usr/bin/passwd carl).[11]

Sudo also logs when a user can't run a command.

Aug 27 23:35:25 pestilence sudo: mike : command not allowed ; TTY=pts/1 ; PWD=/home/mike ; USER=root ; COMMAND=/usr/bin/passwd root

Note the string command not allowed. Looks like someone's trying to escape the cage in his cubicle. Again. The boss needs to have a word with him. Again.

Customizing Sudo Syslog

The default configuration has some weaknesses, though: the log file's location on the local system, and the logs even existing on the local system at all.

On most Unix-like systems, sudo logs get dumped into the main system log, along with the logs from all the other programs running on the machine. This makes the logs more complicated to search than they need to be. Also, successes and failures are logged together. You need both sorts of log messages, but you don't want them simultaneously. Creating one log for successes and one for failures will simplify troubleshooting.

Sudo uses the LOCAL2 log facility by default. Successful sudo runs get priority notice, while unsuccessful ones get the higher priority alert. This means you can easily split the two types of sudo responses into separate log files. Here's how you would do this on a system running traditional syslogd.

```
local2.=notice /var/log/sudo
```

```
local2.=alert /var/log/sudofail
```

Touch the two files and restart syslogd. Logs of successful sudo use go to */var/log/sudo*, while unsuccessful sudo attempts go to */var/log/sudofail*.

You can change the log's facility and the priorities using the options syslog, syslog_badpri, and syslog_goodpri. This lets you avoid conflicts with other software that uses sudo's default priorities and adjust the priorities to accommodate any log monitoring software you might have. Here's a sudoers policy for custom logging.

```
Defaults syslog=local6, syslog_badpri=crit, \
    syslog_goodpri=info
```

Most syslogd implementations let you split out logs by program name as well.

Separating out the sudo log opens up some interesting customer service possibilities. Repeated sudo failures are evidence of a problem. Either a user is testing their limits, or they're trying to do their job but failing, or they're flailing around helplessly. Now you can have a helpdesk flunky pick up the phone and say "Hey, we see

you're having trouble." The end user will either feel like you are watching out for them, or you're watching them very closely. Either way, a little bit of omniscience never hurts your reputation.

Syslog Security Problems

Almost all syslog implementations write logs to the local machine by default. This is a problem for sudo, because a user might alter the log files. If a sysadmin wants to see what her users do on her machines, she need to automatically log to remote machines. This copy must happen in real time. Have syslog send all log messages to a central logging host. This syslog.conf entry for standard syslogd sends all messages to a host called **loghost.**

```
*.* @loghost
```

If you can't send all the system logs, at least send the sudo logs.

```
local2.=notice /var/log/sudo,@loghost
local2.=alert /var/log/sudofail,@loghost
```

Finally, use a syslog daemon that securely transmits messages to your logging host. Programs such as syslog-ng and rsyslog let

you transmit logs encrypted via SSL and/or transport the logs via TCP.

Sudo and Email

Sudo normally sends email to the system's **root** account whenever a user tries to use sudo but fails. You can adjust when sudo notifies you of events, or whether it notifies you at all, with the options mail_always, mail_badpass, mail_no_host, mail_no_perms, and mail_no_user. These notifications can quickly alert when a user is having trouble with sudo. They can also help find intruders – after all, if your web server user starts trying to use sudo, you want to know as soon as possible!

A standard sudo install emails **root** whenever a user has a problem with sudo permissions, either trying to run a command they don't have rights to or if they don't appear in the security policy. If nobody reads emails addressed to root on the local system, those emails will pile up and eventually fill your hard disk. Either forward sudo emails to an account where someone will read them, or disable email notifications.

The mail_no_user flag tells sudo to send an email notification

whenever a user who doesn't appear in the sudo policy attempts to run sudo. Sudo normally enables this option by default, and you've probably seen this email before.

The mail_no_perms option sends a notification whenever a user tries to run a command that they aren't permitted access to. I find this notification useful to quickly find users who are struggling to perform routine tasks with sudo.

Do you want to know when users have password trouble? Use the mail_badpass option to send email whenever a user enters an incorrect password. I find this generates too many messages that don't require any action.

Maybe a user is listed in the sudoers file, but doesn't have access to sudo on this particular host. The mail_no_host option tells sudo to send an email whenever a user tries to use sudo but doesn't have sudo access on that host.

The mail_always option sends an email any time anyone uses sudo, successfully or not. You might want this for testing, but certainly

not in production.

Sudo Debugging

Sometimes sudo can drive you to the brink of madness. Writing a policy is simple enough. Running the sudo command is pretty easy. But things don't always work as you expect. While it's conceivable that you've discovered a legitimate sudo bug, the truth is that you probably don't really understand how sudo interprets your policy.

Debugging lets you watch sudo as it processes your policy. You can see exactly how sudo makes decisions and adjust your sudoers policy to work the way you desire. Configure sudo logging in */etc/sudo.conf*.

Sudo Subsystems and Levels

If you've configured syslog, sudo logging should look very familiar. Log messages are divided into *levels* and *subsystems*.

A level is a measure of severity or priority. The lowest level, debug, includes every trivial bit of crud that passes through sudo. The highest level, crit, only includes problems that keep sudo from

running correctly. In order from least to most severe, the levels are: *debug*, *trace*, *info*, *diag*, *notice*, *warn*, *err*, and *crit*. Which level do you need? That depends on how much detail you want. I find that notice level is enough to identify most problems. The debug and trace levels produce hundreds of lines of output even for simple commands like `sudo –l`, but are very useful when reporting sudo problems to the mailing list. Like syslog, setting a sudo debug level will log everything of the stated priority or higher. If you choose to log notice level events, you get notice, warn, err, and crit levels.

In addition to severity levels, sudo logs via subsystems. You can log activity from each subsystem separately. If you have a problem with sudoedit, you can specifically log only sudoedit events. If sudo seems to match the wrong per-host rules, you can log network interface handling in both sudo and the sudoers policy.

The sudo command logs from the following subsystems:

`args` – command argument processing

conv – user conversation
edit – sudoedit
exec – command execution
main – sudo main function
pcomm – communication with the plugin
plugin – plugin configuration
selinux – SELinux-specific events
utmp – utmp handling

Sudoers policy processing has the following subsystems:

alias – processing for all aliases
audit – BSM and Linux audit code
auth – user authentication
defaults – sudoers Defaults settings
env – environment handling
ldap – LDAP handling
logging – logging events
match – matching users, groups, hosts, and netgroups

nss – network service switch handling

parser – sudoers file parsing

perms – permissions processing

plugin – plugin main function

rbtree – redblack tree internals

Both sudo and the sudo policy plugin share these following subsystems:

All – log everything from everywhere

netif – network interface handling

pty – pseudo-tty related events

util – utility functions

Not sure what subsystem to log? Start with All and trim down from there.

Configuring Debug Logging

Configure logging in *sudo.conf*. The entry needs four parts: the Debug statement, the program or plugin to be debugged, the log

file location, and the subsystems and levels to be logged.

```
Debug sudo /var/log/sudo_debug all@notice
```

The `Debug sudo` statement applies to both the sudo program and the sudoers policy. This example logs to the file */var/log/sudo_debug*. We specifically log all subsystems, at notice level and above.

You can log different subsystems at different levels. If you are experimenting with sudo's authentication system, you might want to crank up authentication logging.

```
Debug sudo /var/log/sudo_debug all@notice, auth@debug
```

You can only have one Debug statement per program or plugin. This means you only get one log file for standard sudo debugging, as the sudo program and the sudoers policy share the sudo Debug statement. If you are using a different policy plugin, it can have its own Debug statement.

Debugging LDAP

One of the common uses of the debugging log is to figure out how sudo is interacting with LDAP (see Chapter 11). Originally you

configured LDAP debugging in *ldap.conf*, but that put the debugging output in the user's window whenever they ran sudo. That was scary. It's now part of the sudo logging system.

To log basic LDAP interactions, log the ldap subsystem. Basic debugging is available at info level, while detailed logging lives at debug.

```
Debug sudo /var/log/sudo_debug all@notice,ldap@info
```

Sudo will now record its LDAP-related activity in the debug log.

Debug Usefulness

Sudo has a lot of subsystems. Some of them, like LDAP and environment purging, produce very useful logs for systems administrators trying to understand what sudo is doing. Others, like the main routine, produce output meaningful only to people who program sudo. If you're trying to understand a weird sudo behavior and you can't see anything useful in the log, increase the number of subsystems you're logging and/or the log level. Worst case,

logging everything at the debug level will get you all the information sudo produces. After that, you'll have to fall back to programs like truss or strace and the sudo-users mailing list.

Sudoreplay

Sudo uses syslog to record user activity. We can debug sudo and create a sudo program log file. But what about detail on what people did within complicated privileged sessions? What if they fired up an interactive system administration tool like sadm or plain old /bin/sh? Enter sudoreplay.

The sudo process is the parent of any command run under sudo. This means that the sudo process can see any input or output of that command. Sudo can log the input and output, give it a timestamp, and display it exactly as it happened.

Enable output logging with the log_output option. Do not log the output from sudoreplay itself, as you'll quickly fill your disk with log messages. And logging output from the reboot and shutdown commands can delay the system's shutdown and recovery, as sudoreplay tries to log any shutdown messages on disk that's just been unmounted as part of the reboot process.

Defaults log_output

```
Defaults!/usr/bin/sudoreplay !log_output
Defaults!/sbin/reboot !log_output
```

The default directory for sudo logging is */var/log/sudo-io*, but you can change this with the iolog option.

You can also enable input logging with the log_input option. This is more problematic as input might contain passwords or other sensitive information. The log_input option only logs what's echoed back to the user, but some programs print sensitive information. If the user's input does not appear in the terminal window, then sudo's input log won't store it.

```
Defaults log_input
```

Most of the time, output logging suffices to see exactly what a user did. If you need input logging, it's available.

You can enable and disable input and output logging on a per-command basis with the LOG_INPUT, NOLOG_INPUT, LOG_OUTPUT, AND NOLOG_OUTPUT tags. If you want to log how users apply certain commands, use these tags in the command-specific rules.

Listing Logged Sudo Sessions

Enable I/O logging on your test machine and run a few commands under sudo to create some logs. Use the sudoreplay list mode (–l) as **root** to view the list of logged sessions.

```
# sudoreplay -l
Sep 1 19:53:42 2013 : mike : TTY=/dev/pts/1 ; CWD=/usr/home/thea ; USER=root ;
TSID=000001 ; COMMAND=/usr/bin/passwd
Sep 1 20:04:42 2013 : thea : TTY=/dev/pts/2 ; CWD=/usr/home/thea ; USER=root ;
TSID=000002 ; COMMAND=/usr/local/bin/emacs /etc/rc.conf
…
```

Each log entry includes several fields, delimited by either colons or semicolons. We start with the full date, in local time. Our first log entry was recorded at 19:53:42, or 7:53 PM, on 1 September 2013.

The next field is the user who ran the command – in the first entry **mike**, and in the second, **thea**.

Then there's the terminal. Sudo runs logged sessions in a new pseudoterminal, so it can capture all input and/or output.

The working directory is next. Editing the copy of */etc/fstab* in your home directory is very different from editing the actual */etc/fstab*, and this field lets you differentiate between those.

The USER field gives the user the command was run as. Here, both Thea and I ran a command as **root.**

The TSID is the name of sudo's log entry. If you want to view the actual session, you'll need this number. When sudo I/O logging is enabled, sudo also adds the TSID to the syslog message.

Finally, the COMMAND is the actual command run. For the first command, I ran passwd, while in the second Thea edited */etc/rc.conf*. Sudo logs the full path to all commands it runs.

Viewing Sessions

To view an actual session, give sudoreplay the TSID of the session in question. In that first session, did I really run passwd to change the root password?

```
# sudoreplay 000001
Replaying sudo session: /usr/bin/passwd
```

```
Changing local password for root
New Password:
Retype New Password:
#
```

Yep, I changed the root password.

Sudoreplay shows sessions in real time, exactly as they happened. If I waited a few seconds to type a password, the replay session pauses exactly there. The replay also appears to pause while I typed the password – there's no visible change because the terminal didn't display any output as I typed the new password.

Altering Playback

The ability to play back sessions is useful, but sometimes a session runs too quickly to make sense or too slowly to watch comfortably.

To interactively change the replay speed on longer sessions, you might want to suspend, slow down, or accelerate playback speed. Use the space bar to pause a replay, and any key to resume. A less than symbol (<) reduces replay speed by half, while a greater than

symbol (>) doubles it.

If you know before starting the replay that you want to adjust the replay speed, preemptively adjust the replay speed with the –m and –s command-line arguments.

The –m flag sets a maximum number of seconds to pause between changes, either key presses or screen output. Maybe you've logged the output of a complicated install process that took a long time to run, and you want to review it with two seconds between each screen update. Or maybe Thea knew from the first time she saw the replay that I spent a lot of time sitting at the password prompt when I changed the root password without authorization, and she wants to speed up the display during yet another Human Resources[12] meeting.

```
$ sudoreplay –m 1
```

Use the –s flag to change the speed of the entire replay. The replay speed is divided by whatever value you give. If you use -s 4, the replay runs four times as fast. If you use –s 0.25, the replay runs

at one-quarter speed.

```
$ sudoreplay –s 2
```

Between –s and -m, and with the interactive controls, you can adjust the replay speed as needed for any situation.

Searching Sudoreplay Logs

Traditionally, you figured out who did what by using grep on the default system log. Sudoreplay's list mode also lets you search by command, user, RunAs, terminal, and more.

The command keyword searches for a command that matches your search term. If your operating system supports POSIX regular expressions, your search term is treated as a regular expression. Otherwise, it's a substring match. Here I search for the passwd command in the sudoreplay logs:

```
# sudoreplay -l command passwd
```

The cwd keyword tells sudoreplay to look for commands run in the given directory. Here I search for all sudo runs in the */etc* directory:

sudoreplay -l cwd /etc

Don't include a trailing slash on the directory name. Also, the directory name must match exactly –searching for */etc* will not match */etc/ssh*. Remember that users don't have to run commands from a directory to affect files in that directory – you can run vi /var/log/messages from their home directory rather than going into the /var/log/ directory and running vi messages.

To search for all sudo sessions run by a specific user, use the account name and the user keyword.

sudoreplay –l user mike

The group keyword searches for commands run as a particular group. The user must have explicitly requested to run a command as this group (i.e., with sudo –g) for this filter to match.

sudoreplay –l group operator

To search for commands run as a specific user, use the runas keyword. Sudo runs commands as root by default, so searching for root would probably get you a lot of results.

sudoreplay –l runas postgres

You can even search by terminal device name with the tty keyword. Want to know who ran sudo on the console? Use the tty keyword, but don't use /dev/ in front of the device name.

sudoreplay –l tty console

One popular way to search logs is by date and time. Sudoreplay has many ways to filter log searches by time, and I cover the most commonly used here. If you're interested in the full details, check the sudoreplay manual page, but any program that lets you search by fortnight contains more search options than any sane person needs. It supports many vernacular time expressions such as "last week," "today," "4 hours ago," as well as dates and times.

To search for all sudo usage on or after a given date, use the fromdate keyword.

sudoreplay -l fromdate "last week"

You must quote multi-word date search terms.

To view all sudo usage before but not including a given date, use the todate keyword.

sudoreplay –l todate today

For search words like today, last week, a fortnight ago, and so on, sudoreplay assumes that the day starts at midnight.

Other popular time formats include exact dates and times with AM or PM. Here we search for what happened between 8PM and 11:59 PM on the first of September, 2013.

```
# sudoreplay -l fromdate "8pm 1 Sep 2013" todate "11:59pm 1 sep 2013"
```

When you use words for months, the day and month can appear in any order. If you use numerical months, the month must appear first. If you drop the year, sudoreplay assumes that it's the current year. This next example searches for any entries after 4 September.

```
# sudoreplay -l fromdate "9/4"
```

Use "4/9" instead, and you'll get matches from 9 April. I avoid confusion by naming months.

You can combine search keywords beyond just dates. The example below searches for my account running sudo after the first of September.

```
# sudoreplay -l fromdate "9/1" user mike
```

Combine searches with the or operator.

sudoreplay -l command /bin/sh or command /bin/bash

If you need to group different search terms, parentheses can help.

sudoreplay –l (command /bin/sh or command /bin/bash) user mike

Fortunately I use tcsh, so this won't catch me.

This should get you well on your way to searching your I/O logs. I recommend not drinking anything when you first peruse what your users actually run through sudo, as a spit-take wastes good caffeine.

Sudoreplay Risks

Sudoreplay is a powerful addition to a system administrator's toolkit, but it does have problems. If you log session input, you might capture sensitive data such as passwords in the sudo logs. Those logs are unencrypted, and a troublesome user who can weasel himself into root-level access could find that information.

The sudoreplay logs themselves are stored on the local system. An unauthorized user could damage, alter, or delete those logs. As

I write this sudo cannot store its I/O logs on a remote system, but session logging is a fairly new feature. I expect that someone will create a solution for off-server session log storage before long. The good news is that sudoreplay logs are much harder to edit than a text log file. While the I/O log certainly isn't tamper-proof, unskilled tampering will be quite obvious.

Chapter 13: Authentication

Sudo's authentication system looks pretty straightforward: enter your password and run a privileged command. But sudo will let you change how it handles your password, how often you must enter your password, and if it takes a password at all. You can tell sudo to demand stronger authentication than a password by requiring, say, an SSH agent or a hardware token or some other authentication method I've never even heard of, and how it handles to authentication methods.

We'll start with the simplest case, password management.

Sudo Password Configuration

You can control how sudo requests passwords, how many times sudo lets the user try to enter a password, and how sudo shares authentication between terminal sessions.

Password Attempts and Timeouts

Sudo gives users three chances to enter their password. Maybe your users can't successfully type their passwords on the first, second, or third try. Use the password_tries option to give them a few extra attempts.

Sudo gives a user five minutes to type their password before timing them out. I, for one, find this excessive. If a user can't type their password in sixty seconds, I don't want them on my server. Sadly, Thea is a more accommodating soul than myself. Use the passwd_timeout option to set a timeout in minutes.

```
Defaults passwd_tries=5, passwd_timeout=2
```

Users have five tries to enter their password, but their password

prompt times out in two minutes.

Sudo normally doesn’t give any feedback when a user enters a password. If you want the user to see something when they type, use the pwfeedback option.

```
$ sudo -l
Password:*********
```

Most security people discourage using the pwfeedback option. Anyone watching the user type learns the length of the user's password.

Target Password

One of sudo's features is that it demands the user's password to perform privileged actions, rather than the root password. In certain environments the system owner might want the user to enter the target user's password rather than their own – usually for audit compliance reasons, in my experience. Use the rootpw, targetpw, and runaspw options for this.

The rootpw option tells sudo to require the root password rather

than the user's password. Here, users in the wheel group must use the root password for sudo.

```
Defaults:%wheel rootpw
```

The targetpw option tells sudo to require the target user's password rather than the user's password. If the user uses the –u command-line argument to run a command as another user, he needs to enter that user's password.

```
Defaults targetpw
```

Finally, the runaspw option tells sudo to require the password of the default RunAs user instead of the user's password. You might want users who run any programs in the Oracle directory to use the **oracle** account's password rather than their own.

```
Defaults>oracle runaspw
```

Between all of these, you can customize the necessary password however you want. You do risk confusing the user, however. If only there was some way to tell the user which password they needed to enter…

Customizing the Password Prompt

Sudo's password prompt is kind of boring. `Password:` does the job, but the `passprompt` option lets you do more interesting things.

```
Defaults passprompt "Your wussy password is:"
```

This is mildly amusing at best. But using escape characters in the password prompt string makes the custom prompt useful.

To use the machine's hostname in the password prompt, use `%H` or `%h`. `%h` is the short hostname, while `%H` is the fully qualified hostname. Sudo can only get the fully qualified hostname if the `fqdn` option is set or the `hostname` command returns the fully qualified hostname.

```
Defaults passprompt="Your password on %h is:"
```

To name the user whose password sudo expects, use `%p`. This reminds users what password to enter when you're using the `rootpw`, `runaspw`, and `targetpw` options.

```
Defaults passprompt="Enter %p's password:"
```

To name the user who the command will run as, use `%U`. If your users frequently run commands as users other than root, this can

help them keep things straight. Heck, it helps *me* keep things straight.

```
Defaults passprompt="Enter %p's password to run command as %U:"
```

To name the user running sudo, use `%u`. If you have multiple user accounts, this might also help you keep them straight.

```
Defaults passprompt="%u: enter %p's password to run command as %U:"
```

If you need a percent sign in your prompt, use two consecutive percent signs (`%%`).

The `passprompt` option expects that the system's authentication system (PAM or similar) uses a password prompt of `Password:`. If your system uses something else as a password prompt, use the option `passprompt_override` to stop that check and insist that sudo use your custom prompt.

Authentication Caching and Timeout

Sudo doesn't cache your password or other authentication credentials. It does remember the date and time that you last successfully authenticated in a given terminal session, however.

This lets you run sudo again within the next few minutes without using a password. You can control how sudo treats this cache and how long sudo will run commands for you without re-entering your password. If you run `sudo –V` as root and search for the string `timestamp`, you'll see sudo's authentication timing settings.

```
# sudo -V | grep timestamp
Authentication timestamp timeout: 5.0 minutes
Path to authentication timestamp dir: /var/db/sudo
```

Once you enter your password, you won't need to enter it again for five minutes in that terminal window. Change this with the `timestamp_timeout` option and a number of minutes for the timeout. Use a 0 to disable the timestamp.

```
Defaults timestamp_timeout=0
```

If you use a negative value, the timestamp will never expire. Don't do that.

According to `sudo –V`, the timestamps are in the */var/db/sudo* directory. Change the directory with the `timestamp_dir` option. While root normally owns the directory and the timestamps in it, you

could change this with the timestamp_owner option. I *strongly* recommend that you leave these settings at your operating system defaults unless your operating system vendor or the sudo developers tell you to change them.

User Updating Authentication Timeouts

Users can interact with the authentication cache by either updating the time they last authenticated or by eradicating the cached credentials.

If you want to authenticate to sudo without running any commands, run sudo –v. Sudo will prompt you for your password, verify it, and update the timestamp. Use this when you're about to run a whole bunch of commands via sudo and don't want to get stopped by a password prompt halfway through.

If you want sudo to ignore your authentication timestamp cache for this terminal window, use the –k option. Used on its own, it invalidates the authentication timestamp. If you specifically want

sudo to request authentication the next time you run a command, add –k to the command line.

```
# sudo –k ifconfig
```

Even if you have time left in your authentication timestamp, sudo will now ask you to authenticate.

To totally remove the authentication timestamp from all of your sudo sessions, run sudo –K. This entirely removes your timestamps, or if it can't remove them, resets them to 31 December 1969. Use sudo –K before walking away from your computer, even if you run a screen locking program. Remember, a system administrator can overcome most screen locks. You don't want a cretin like me unlocking your workstation and using your sudo access.

Disabling Authentication

Sometimes you want a user to have the ability to run a command without entering a password. If you're always reconfiguring your laptop to connect to different networks, it might make sense to not bother with a password for dhclient, ifconfig, and related commands. You might even want the ability to always run sudo without a password on your desktop. And running sudo without a password is very reasonable for automated tasks.

Broadly disabling authentication for sudo is unwise. Yes, it's most convenient. Also, any application that gains control of your user session will have total access to all of your sudo privileges. If you're running an operating system like Ubuntu, which gives the initial user full root access via sudo, then the rogue process will completely own your machine. Disabling sudo authentication is equivalent to deliberately implementing the Windows 95 security system.[13] If you don't want to bother entering a password when

you need sudo, look at an alternate authentication mechanism such as an SSH agent (see *Sudo and PAM* later this chapter).

My examples assume you selectively disable authentication. You can extrapolate them to globally disable authentication or look in the default sudoers file shipped with most operating systems.

The authenticate Option

One way to control authentication is the authenticate option on Defaults statements. The authenticate option doesn't appear in most sudoers files, because it's an invisible global default. Negate it to disable authentication. Here I disable authentication for ifconfig and dhclient:

```
Defaults!/sbin/ifconfig,/sbin/dhclient !authenticate
```

I can now set up my laptop at the coffee shop without bothering with my password.

Authentication Tags

If you want to very precisely control authentication in your sudoers

policy, use the tags PASSWD and NOPASSWD on specific sudoers rules. you rarely see the PASSWD tag, as it's the default. Use NOPASSWD to turn off the password requirement.

```
pete dbtest1 = (oracle) NOPASSWD: /opt/oracle/bin/*
```

Pete may use sudo to run any Oracle command as the user **oracle** on the host **dbtest1** without entering a password.

Sharing Authentication Between Sessions

Sudo normally includes the terminal device in the authentication timestamp. That is, sudo not only uses the username but also the terminal device (or TTY) to identify a user session's sudo authentication.

Assume I SSH into a server twice, and my sessions use virtual terminals */dev/vty2* and */dev/vty3*. If I use sudo in the *vty2* terminal window, and want to use it again in the *vty3* window, I must authenticate in the *vty3* window.

Some operating systems include a sudo package configured to permit sharing sudo authentications between terminal sessions. If

you open two SSH sessions to a server and authenticate via sudo in one session, the other session can use that same authentication timestamp.

This seems strange to many people – it certainly struck me as odd. But it's really hard to isolate the two processes from each other when they're both owned by the same user. Users have complete control over their own processses, after all. This means that if a skilled intruder penetrates a user account while the user is active in another session, the intruder can use tools like `ptrace` and `gdb` to run commands via sudo as long as any terminal session has a valid timestamp. Still, requiring separate authentication for each terminal window does increase the skill an attacker needs to further penetrate the system – your average script kiddie won't have the expertise needed to hijack another terminal's sudo session.

You can control per-terminal authentication with the `ttytickets` option. Negating this option lets multiple terminals share a single timestamp for authentication. Unless you have very specific

reasons for disabling per-terminal timestamps, however, I strongly encourage you to leave it on.

Querying Sudo

Sudo has two user functions that don't run commands. The –l flag tells sudo to print out the user's sudo policy, so the user can see what they have access to. The –v flag updates the user's authentication timeout. Users must enter their password to use these functions, but you can change these features to only require a password under certain conditions.

The listpw option controls whether user must enter his password to list his access, while the verifypw option controls whether a user must enter his password to update his authentication timestamp. Each of these options can have one of four values: *any*, *always*, *all*, and *never*.

The default, any, means that if any of the user's sudoers rules have NOPASSWD or !authenticate set, the user doesn't need to enter a password to use the function. Turning off password authentication for one command means enabling passwordless use of sudo's –l and –v flags.

If these options are set to always, the user must enter a password every time they want to use these functions. Even if the user's authentication timestamp has not expired, the user must always enter a password to use –l or –v. always means "always."

If you set these options to all, -l and –v will request a password unless the user has passwordless access to all of their permitted commands on this host. They don't need passwordless access to all possible commands, mind you, only passwordless access to all of the commands that they can run.

The never setting means that users are never asked for a password to use –l or –v.

Here sudoers tells sudo –l to demand a password for every user except Thea. We also disable asking for a password to update the authentication timestamp on the host **www**.

```
Defaults listpw=always
Defaults:thea listpw=never
Defaults@www verifypw=never
```

Changing the listpw and verifypw options for commands or RunAs

doesn't make much sense, but you can sensibly change them for hosts and users.

Lecture

I used the sudo lecture for many examples in Chapter 5, but let's give it more concentrated treatment. The "lecture" is the message displayed when you first authenticate to sudo.

```
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.
```

You then get a chance to enter your password. This is a nice general warning, but the `lecture` and `lecture_file` options give you a chance to give more specific lectures as the situation demands.

The lecture option accepts three values. The default, `once`, tells sudo to give each user the lecture once and never again. Using `always` tells sudo to always lecture the user, while using `never` or `!lecture` disables the lecture entirely.

Use the `lecture_file` option to give a file containing your own

lecture. You can set the lecture based on any default setting. So Thea might set this configuration:

Defaults:mike lecture=always,
lecture_file=/etc/sudo/mike-lecture

The *mike-lecture* file might contain something like this.

Everything you do is logged.

And Thea studies the logs.

I'm on to you, mister.

Apparently someone thinks I'm trouble.

Sudo and PAM

Passwords aren't very strong authentication tokens. Most users create terrible passwords, and a sufficiently persistent intruder can eventually guess even decent passwords. Adding another layer of authentication to your privileged processes, or eliminating passwords altogether, can improve your security.

Pluggable Authentication Modules, or PAM, permit system administrators to attach new authentication systems to programs. Each authentication system comes in a *PAM module*, containing the code needed to use the authentication system. In addition to the usual password, Unix, Kerberos, and LDAP modules found on Unix-like systems, you can install PAM modules that implement Google Authentication, RSA tokens, Windows SMB authentication, and many more. Not all operating systems support PAM, but if yours does, you can leverage PAM to authenticate sudo.

Just as this is not a book on LDAP, this is not a book on PAM.

This section probably contains enough knowledge to get my example PAM module working on your system with sudo, but it won't make you into a PAM wizard. And don't forget that many vendors have their own PAM system, while the open source world has two similar but not identical implementations. If you want an advanced PAM configuration, check your operating system's documentation to see what you have and what it can do.

Lots of the PAM modules aren't suitable for my environment, however. Using Google Authenticator not only removes the source of trust from my network, it means that if my external network connection fails I cannot authenticate. I will not authenticate against a Windows domain or deploy RSA tokens in this company. The SSH agent authentication module, however, is interesting.

An SSH agent runs on the user's desktop computer. It holds a user's decrypted SSH authentication keys in memory. If the SSH client or session needs to validate posession of the keys, it asks the desktop agent to perform the validation. This is stronger than

password authentication, as the user must have both the key and the passphrase for the key. Of course, you shouldn't allow all SSH servers access to your agent, but that's pretty easily configured. If this paragraph made no sense to you, permit me to recommend my book *SSH Mastery* (Tilted Windmill Press, 2012).

The PAM module pam_ssh_agent_auth (http://pamsshagentauth.sourceforge.net/) permits processes to authenticate against your SSH agent. I'll use this module as an example of adding security systems to sudo.

Prerequisites

Before configuring sudo to use SSH agent authentication, check that you have all the prerequisites.

You must have user *authorized_keys* files on the local machine. This means that if you're using an SSH server that gets its keys from LDAP or another external source, you cannot use pam_ssh_agent_auth.

Your SSH client must forward your desktop SSH agent to the server, and the server must accept the agent forwarding. To see if this works in your SSH session, check for the environment variable SSH_AUTH_SOCK.

```
$ echo $SSH_AUTH_SOCK
/tmp/ssh-u2ThOMa9py/agent.24047
```

If this variable contains a path, either your agent forwarding works or you have a truly bizarre problem. If this variable doesn't exist, check your SSH client and server settings.

Now install pam_ssh_agent_auth. Unlike much modern software, pam_ssh_agent_auth doesn't have all kinds of fancy configuration options. If your operating system has a packaged version – and it probably does – use it.

SSH agent authentication needs the environment variable SSH_AUTH_SOCK, which SSH automatically sets to point to a local socket connect to your SSH agent. You need to permit this environment variable in your sudoers policy. I recommend also passing SSH_CLIENT, SSH_TTY, and SSH_CONNECTION so

that programs like sftp work.

Sudo defaults to setting the authentication timestamp when you authenticate. This behavior will drive you buggy when trying to deploy a new authentication system. Disable the timestamp by setting the option timestamp_timeout to 0.

```
Defaults env_keep += "SSH_CLIENT SSH_CONNECTION SSH_TTY
SSH_AUTH_SOCK",
    timestamp_timeout=0
```

Once these prerequisites work you can proceed to configuring the PAM module.

Configuring PAM

PAM keeps authentication configurations in system directories such as */etc/pam.d* or */usr/local/etc/pam.d*. A PAM-aware program searches for its PAM in these directories. Check these directories for a file named *sudo*.

PAM policies include four different types of rules: *auth*, *account*, *session*, and *password*. Changing authentication methods

requires changing the auth rules. Not all PAM policies have all rule types – many policies don't have password rules. Each rule calls a PAM module such as pam_unix, pam_ldap, pam_mkhomedir, and so on.

The PAM module pam_unix handles traditional password authentication. Find an authentication rule in sudo's PAM configuration somewhat like this one.

```
auth required pam_unix.so no_warn try_first_pass nullok
```

This rule tells sudo to use passwords for authentication. To use SSH agent authentication instead of passwords, replace the password rule with your own.

```
auth sufficient pam_ssh_agent_auth.so file=~/.ssh/authorized_keys
```

What does this mean? Authenticating with the method in the shared library pam_ssh_agent_auth.so is sufficient to log on to the system. The file= text gives the path to the user's *authorized_keys* file, which is commonly in *$HOME/.ssh/authorized_keys*. You might need to give the full path to *pam_ssh_agent_auth.so*, depending on how your operating system installs new PAM libraries and how your PAM implementation finds them.

Save your changes to the sudo PAM policy. You should now be able to authenticate to sudo with your SSH agent. Flush your authentication timestamp (if any) and try it.

```
$ sudo –K
$ sudo touch /tmp/test
```

While my PAM rule works for the most common case, a server can store its *authorized_keys* files in several ways. The pam_ssh_agent_auth library must know where the keys are and the acceptable permissions on the key files.

authorized_keys Permissions

In the simplest case, a user owns their own *authorized_keys* file. Some environments don't let users change their own *authorized_keys*, however. Instead, key file updates go through a central management system which copies them to the target host. In such an environment, a compromised user cannot change the key files on the server. The question becomes: who owns the key files?

The allow_user_owned_authorized_keys_file option tells

pam_ssh_agent_auth that the user can own the *authorized_keys* file. This option activates automatically when the key file is in the user's home directory.

Without this option set, and if the *authorized_keys* file is not in the user's home directory, pam_ssh_agent_auth expects **root** to own the key file. If the file is not owned by **root**, authentication fails.

authorized_keys Location

While most tutorials tell you to put *authorized_keys* in the user's *$HOME/.ssh* directory, many organizations use other standards. You must tell pam_ssh_agent_auth where to find the files. The module includes several escape characters for this purpose.

The tilde (~) and %h characters represent the user's home directory.

%H represents the short hostname (without the domain name), while %f means the fully qualified hostname.

Finally, %u represents the username.

Suppose you stored your keys in */etc/sshkeys/*, where each user has a file named after their username. These key files are owned by **root.**

```
auth sufficient pam_ssh_agent_auth.so file=/etc/sshkeys/%u
```

If users can write their own key files in this directory, you must add the allow_user_owned_authorized_keys_file option at the end of the PAM rule.

Debugging pam_ssh_agent_auth

If sudo prompts you for a password and waits for you to do so, you haven't removed the password policy. If sudo prompts you for a password three times in a row without waiting for you to enter the password, and then displays a failure message, sudo is using the PAM module but cannot connect to your SSH agent. Check your agent forwarding. If you still have problems, configure logging in *sudo.conf* to see where things break.

Once you get pam_ssh_agent_auth working with sudo, you can further expand authentication requirements. You want to require an

SSH agent, a password, *and* Google Authentication? You can do it. It's kind of daft, but you can do it.

And given this, you can now make sudo do anything you want.

Afterword

You should now know more about sudo than the vast majority of people who didn't write it. Congratulations! But there's more to learn. If you have a weird sudo problem, check the sudo web site at http://sudo.ws, the sudo man pages, and the archives of the sudo-users mailing list. Sudo has been successfully deployed on millions of very different systems, and it can work for you too.

Always be aware that sudo might not fit your organization, however. Some applications expect to own the server, and trying to restrict those applications is futile at best. If you manage your organization by running shell scripts as root, running those same shell scripts with sudo will leave lots of ways for unauthorized users to escalate their privileges. Sudo is useful, but a sysadmin who understands when a specific tool won't solve his problem is more useful.

And the next time someone tells you that "Sudo is how you get

root," treat them to a short sharp visit from the Slap Fairy.

---

[1] If you haven't played with Ansible (http://ansible.cc), you really should.

[2] I have very few unbreakable rules for being a "real" sysadmin. One of them is, real sysadmins *can* use vi. Vi and ed are the two editors you can be confident of finding on any Unix-like system. "Can't use vi" means "not a sysadmin."

[3] I recently learned that the ipset command uses -! as a common argument. Presumably the developers were out of letters and numbers, and when they run out of symbols they'll proceed to blood samples.

[4] But if I tell her what happened to her comfy chair, I'll never get access to anything ever again.

[5] I said that with a straight face? Wow.

[6] Running a shell that can't execute commands is an education. Try it sometime.

[7] This is also known as "Management won't let me do my job" Syndrome, which is not improved by developing "I gave them an excuse to fire me" Disorder.

[8] If an Evil Secret Agency with access to Super Top Secret Digest Cracking

Hardware™ wants to compromise your computer, he won't bother replacing binaries with treacherous versions carefully engineered to have the same checksum. He'll use your kneecaps. And a hammer.

[9] I know you have a procedure for installing servers. After years in this business, I am firmly convinced that no human being is capable of installing two servers identically.

[10] A couple distributions once required blood sacrifices at the second dark of the moon in a month to make sudo read a policy from LDAP, but I'm *assured* that this behavior was corrected after enough users filed sufficiently detailed bug reports.

[11] When Carl wants to know who changed his password, the boss can tell him. And I'll be in trouble again.

[12] I'd probably be in my own meeting with HR a little after, if I wasn't the owner's brother-in-law.

[13] For those readers too young to remember: Windows 95 had no security system.