PEARSON OPEN SOURCE SOFTWARE SERIES

Linux® Hardening in Hostile Networks

Server Security from TLS to TOR

Kyle Rankin

Linux® Hardening in Hostile Networks

Linux® Hardening in Hostile Networks

Server Security from TLS to Tor

Kyle Rankin

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017942009

ISBN-13: 978-0-13-417326-9
ISBN-10: 0-13-417326-0

1 17

❖

This book is dedicated to my wife, Joy,
without whom it would have never been finished.

❖

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Computer and software security has always been an important topic. Today it is also an urgent one; security breaches continue to increase exponentially, and even GNU/Linux systems, which have traditionally been less prone to problems, are subject to devastating attacks. If you're using a GNU/Linux system for anything at all critical—even if it's just your email and home tax accounting—you need to know how to protect it!

This book is the right "go-to" place to get started. Beginning with the basics, the author covers what you need to know for all important areas of GNU/Linux system management, in clear and readable prose. After reading it you'll be able to improve the safety of your systems and be prepared to go further on your own. The book is written in a careful, vendor-neutral manner, so that everything applies as broadly as possible; with any luck this book will be useful to you again and again, instead of becoming obsolete upon the next release of Distro XYZ.

I hope you'll agree with me that this book is worth your time. Keep safe!

—Arnold Robbins
Series Editor

This page intentionally left blank

Preface

We are living in the golden age of computer hacking. So much of our daily lives—from how we communicate, to how we socialize, to how we read news, to how we shop—is conducted on the Internet. Each of those activities rely on servers sitting somewhere on the Internet, and those servers are being targeted constantly. The threats and risks on the Internet today and the impact they can have on the average person are greater than ever before.

While there are exceptions, most computer hackers a few decades ago were motivated primarily by curiosity. If a hacker found a major vulnerability in a popular application, she might break the news at a security conference. If she compromised a network, she would look around a bit and might install a backdoor so she could get back in later, but generally speaking the damage was minimal. These days, many hackers are motivated by profit. A zero-day vulnerability (i.e., a new, unpatched vulnerability not disclosed to the vendor) in a popular application can be sold for tens to hundreds of thousands of dollars. Databases from hacked networks are sold on the black market to aid in identity theft. Important files are encrypted and held for ransom.

Hacker motivations are not the only thing that's changed; so have the hackers themselves. While you will still find pasty, white male hackers wearing black hoodies and hacking away in a basement, that stereotype doesn't match the reality. The spread of high-speed, always-on Internet throughout the world means that Internet users in general, and hackers specifically, reflect the diversity of the world itself. Instead of a black hoodie, a hacker today might wear a dress, a tie, or a uniform, and may work for organized crime or the military. Hackers are international and diverse, and so are their targets.

With everyone online, hacking has become a very important part of surveillance, espionage, and even warfare. Nation-state hackers have become more overt over the years to the point that now it's not uncommon to hear of nation-state actors compromising power grids, nuclear facilities, or major government networks. Nation-state hackers are well-funded, well-trained, and as a result they have sophisticated tools and methods at their disposal. Unlike conventional military tools, however, these tools find their way into the ordinary hacker's toolkit sometimes after only a year or two. This means that even if your threat model doesn't include a nation-state attacker, it must still account for last year's nation-state hacking capabilities.

Hackers aren't the only thing that's different; so are the targets. In the past, hackers might target well-known, large companies, banks, or governments, and they would target them primarily from the outside, spending a lot of time researching the target, discovering vulnerabilities in their software, and then exploiting them. The external

network was viewed as a hostile war zone, the internal network was viewed as a safe haven, and the two were connected by computers in a network actually called a "demilitarized zone" (DMZ). Systems administrators working at a random company would throw up a firewall on the perimeter of their network, install antivirus software on their workstations, and console themselves with the idea that their network isn't interesting enough, and their data isn't valuable enough, to attract a hacker.

Today every computer on the network is a target, and every network is hostile. While you still have hackers who spend a lot of time carefully probing a high-value target, the bulk of the hacking that goes on these days is fully automated. The goal of many hackers is to build the largest collection of compromised machines possible so they can use them to launch further attacks. Those hackers don't necessarily care which computers they compromise, they just scan the Internet attempting to guess SSH passwords or looking for computers with known vulnerabilities so they can automatically exploit them. Each time a new vulnerability is announced in a major piece of software, it only takes a short time before hackers are scanning for it and exploiting it. Once a hacker has a foothold on any machine on your network, whether it's a web server or a workstation, they will automatically start probing and scanning the rest of the internal network for vulnerable machines.

Cloud computing has further eroded the notion of an "internal" and an "external" network. In the past, it would be really difficult for a hacker to buy a server and rack it next to you on your network, yet cloud computing makes this as easy as a few clicks. You have to throw out the assumption that your cloud servers are communicating with each other over a private network and act like every packet is going over a hostile, public network because in many cases it is.

The Good News

Despite all of this, we defenders actually have the advantage! We get to define how our networks look, what defenses we put in place, and if this is a battle, we have control of the battlefield if we choose to take it. With all the talk about sophisticated hackers, the fact is many of the compromises you hear about in the news didn't require sophisticated skills—they could have been prevented by a few simple, modern hardening steps. Time and time again, companies spend a lot of money on security yet skip the simple steps that would actually make them secure. Why?

One of the reasons administrators may not apply modern hardening procedures is that while hacker capabilities continue to progress, many of the official hardening guides out there read as though they were written for Red Hat from 2005. That's because they *were* written for Red Hat in 2005 and updated here and there through the years. I came across one of these guides when I was referring to some official hardening benchmarks for a PCI audit (a Payment Cards Industry certification that's a requirement for organizations that handle credit cards) and realized if others who were new to Linux server administration ran across the same guide, they likely would be overwhelmed with all the obscure steps. Worse, though, they would spend hours performing obscure sysctl tweaks and end up with a computer that was no more protected against a modern attack.

Instead, they could have spent a few minutes performing a few simple hardening steps and ended up with a more secure computer at the end.

For us defenders to realize our advantages, we have to make the most of our time and effort. This book aims to strip away all that outdated information and skip past a lot of the mundane hardening steps that take a lot of time for little benefit. Where possible, I try to favor recommendations that provide the maximum impact for the minimum amount of effort and favor simplicity over complexity. If you want a secure environment, it's important to not just blindly apply hardening steps but to *understand* why those steps are there, what they protect against, what they don't protect against, and how they may apply (or not) to your own environment. Throughout the book, I explain what the threats are, how a particular hardening step protects you, and what its limitations are.

How to Read This Book

The goal of this book is to provide you with a list of practical, *modern* hardening steps that take current threats into account. The first few chapters of the book focus on more general security topics including overall workstation, server, and network hardening. The next few chapters focus on how to harden specific services such as web servers, email, DNS, and databases. Finally, I end the book with a chapter on incident response, just in case. I realize that not everyone has the same level of threat, not everyone has the same amount of time, and not everyone has the same expertise. I've structured every chapter in this book based on that and split each chapter into three main sections. As you progress through each section, the threats and the hardening steps get more advanced. The goal is for you to read through a particular chapter and follow the steps at least up to the point where it meets your expertise and your threat, and hopefully you'll revisit that point in the chapter later, when you are ready to take your hardening to the next level.

Section 1

The first section of every chapter is aimed for every experience level. This section contains hardening steps that are designed for maximum benefit for minimum time spent. The goal is for these steps to only take you a few minutes. These are hardening steps that I consider to be the low bar that everyone should try to meet no matter their level of expertise. They should help protect you from your average hacker out there on the Internet.

Section 2

The second section of each chapter is aimed at hardening steps for intermediate to advanced sysadmins to protect you from intermediate to advanced attackers. While many of the hardening steps get more sophisticated in this section and may take a bit more time to implement, I have still tried to keep things as simple and fast as possible. Ideally, everyone would read at least part of the way into this section and apply some of the hardening steps, no matter their threat model.

Section 3

The third section of each chapter is where I have a bit of fun and go all out with advanced hardening steps aimed at advanced up to nation-state attackers. Some of these hardening steps are rather sophisticated and time-consuming, whereas others are really just the next step up from the intermediate approaches in Section 2. Although these steps are aimed at protecting against advanced threats, remember that today's advanced threats tend to find their way into tomorrow's script kiddie toolkits.

What This Book Covers

Now that we know how the chapters are structured, let's look at what each one covers.

Chapter 1: Overall Security Concepts

Before we get into specific hardening techniques, it's important to build a foundation with the security principles we will apply to all hardening techniques in the rest of the book. No security book can cover every possible type of threat or how to harden every type of application, but if you understand some of the basic concepts behind security you can apply them to whatever application you'd like to secure. Section 1 of Chapter 1 introduces some essential security concepts that you will apply throughout the book and finishes up with a section on choosing secure passwords and general password management. Section 2 elaborates on the security principles in the first section with a focus on more sophisticated attacks and provides a general introduction to two-factor authentication. Section 3 examines how general security principles apply in the face of an advanced attacker and discusses advanced password-cracking techniques.

Chapter 2: Workstation Security

A sysadmin workstation is a high-value target for an attacker or thief because administrators typically have privileged access to all servers in their environments. Chapter 2 covers a series of admin-focused workstation-hardening steps. Section 1 covers basic workstation-hardening techniques including the proper use of lock screens, suspend, and hibernation, and introduces the security-focused Linux distribution Tails as a quick path to a hardened workstation. The section finishes up by covering a few fundamental principles of how to browse the web securely including an introduction to HTTPS, concepts behind cookie security, and how to use a few security-enhancing browser plugins. Section 2 starts with a discussion of disk encryption, BIOS passwords, and other techniques to protect a workstation against theft, a nosy coworker, or a snooping customs official. The section also features more advanced uses of Tails as a high-security replacement for a traditional OS including the use of the persistent disk and the GPG clipboard applet. Section 3 covers advanced techniques such as using the Qubes OS to compartmentalize your different workstation tasks into their own VMs with varying levels of trust. With this in place if, for instance, your untrusted web browser VM gets compromised by visiting a bad website, that compromise won't put the rest of your VMs or your important files at risk.

Chapter 3: Server Security

If someone is going to compromise your server, the most likely attack will either be through a vulnerability in a web application or other service the server hosts, or through SSH. In other chapters, we cover hardening steps for common applications your server may host, so Chapter 3 focuses more on general techniques to secure just about any server you have, whether it's hosting a website, email, DNS, or something completely different. This chapter includes several techniques to harden SSH and covers how to limit the damage an attacker or even a malicious employee can do if he gains access to the server with tools like apparmor and sudo. We also cover disk encryption to protect data at rest and how to set up a remote syslog server to make it more difficult for an attacker to cover her tracks.

Chapter 4: Network

Along with workstation and server hardening, network hardening is a fundamental part of infrastructure security. Section 1 of Chapter 4 provides an overview of network security and then introduces the concept of the man-in-the-middle attack in the context of an attacker on an upstream network. Section 1 finishes up with an introduction to iptables firewall settings. Section 2 covers how to set up a secure private VPN using OpenVPN and how to leverage SSH to tunnel traffic securely when a VPN isn't an option. It then covers how to configure a software load balancer that can both terminate SSL/TLS connections and can initiate new ones downstream. Section 3 focuses on Tor servers, including how to set up a standalone Tor service strictly for internal use, as an external node that routes traffic within Tor, and as an external exit node that accepts traffic from the Internet. It also discusses the creation and use of hidden Tor services and how to set up and use hidden Tor relays for when you need to mask even that you are using Tor itself.

Chapter 5: Web Servers

Chapter 5 focuses on web server security and covers both the Apache and Nginx web servers in all examples. Section 1 covers the fundamentals of web server security including web server permissions and HTTP basic authentication. Section 2 discusses how to configure HTTPS, how to set it as the default by redirecting all HTTP traffic to HTTPS, how to secure HTTPS reverse proxies, and how to enable client certificate authentication. Section 3 discusses more advanced web server hardening including HTTPS forward secrecy and then web application firewalls with ModSecurity.

Chapter 6: Email

Email was one of the first services on the Internet, and it's still relied on by many people not just for communication but also security. Section 1 of Chapter 6 introduces overall email security fundamentals and server hardening, including how to avoid becoming an open relay. Section 2 covers how to require authentication for SMTP relays and how to enable SMTPS. Section 3 covers more advanced email security features that both aid in spam prevention and overall security such as SPF records, DKIM, and DMARC.

Chapter 7: DNS

Domain Name Service (DNS) is one of those fundamental network services to which many people never give a second thought (as long as it's working). In Chapter 7, we cover how to harden any DNS server before you put it on a network. Section 1 describes the fundamentals behind DNS security and how to set up a basic hardened DNS server. Section 2 goes into more advanced DNS features such as rate limiting to help prevent your server from being used in DDOS attacks, query logging to provide forensics data for your environment, and authenticated dynamic DNS. Section 3 provides an introduction to DNSSEC and the new DNSSEC records and discusses how to configure DNSSEC for your domain and how to set up and maintain DNSSEC keys.

Chapter 8: Database

If there is only one place in your infrastructure that holds important information, it's likely to be a database. In Chapter 8, we discuss a number of different approaches to database security for the two most popular open-source database servers: MySQL (MariaDB) and Postgres. Starting with Section 1, we cover some simple security practices you should follow as you set up your database. Section 2 then dives into some intermediate hardening steps including setting up network access control and encrypting traffic with TLS. Section 3 focuses on database encryption and highlights some of the options available for encrypted data storage in MySQL and Postgres.

Chapter 9: Incident Response

Even with the best intentions, practices, and efforts, sometimes an attacker still finds a way in. When that happens, you will want to collect evidence and try to find out how he got in and how to stop it from happening again. Chapter 9 covers how to best respond to a server you suspect is compromised, how to collect evidence, and how to use that evidence to figure out what the attacker did and how he got in. Section 1 lays down some fundamental guidelines for how to approach a compromised machine and safely shut it down so other parties can start an investigation. Section 2 gives an overview on how to perform your own investigation and discusses how to create archival images of a compromised server and how to use common forensics tools including Sleuth Kit and Autopsy to build a file system timeline to identify what the attacker did. Section 3 includes walking through an example investigation and guides to forensics data collection on cloud servers.

Appendix A: Tor

Chapter 4 discusses how to use Tor to protect your anonymity on a network, but it focuses more on how to use Tor and less about how Tor works. Here I dive a bit deeper into how Tor works and how it can protect your anonymity. I also discuss some of the security risks around Tor and how you can mitigate them.

Appendix B: SSL/TLS

Throughout the book, I explain how to protect various services with TLS. Instead of bogging you down with the details of how TLS works in almost every chapter, I've put those details here as a quick reference in case you are curious about how TLS works, how it protects you, its limitations, and some of its security risks and how to mitigate them.

Conventions

This book uses a monospace font for code. Code lines that exceed the width of the printed page are indicated by a continuation character (➥) at the start of the portion of the line that has wrapped to indicate it is all one line.

This page intentionally left blank

Acknowledgments

First, thanks to Aaron, who's been encouraging me for years to shift more of my sysadmin focus over to security (I'm finally starting to listen) and who provided some great feedback on this book. Thanks also to Shawn, Anthony, and Marielle for all their valuable comments on the book.

Thanks to my editor, Debra Williams Cauley, who believed in this book and patiently worked with me to make some ideas we threw around a reality. Thanks specifically to Chris Zahn and also the rest of the Addison-Wesley team for their help in structuring, editing, and laying out this book.

Finally, thanks to all of the defenders out there trying to keep everyone safe and secure. Attackers get the bulk of the press, conference talks, awards, and overall attention in the security community. Like sysadmin work, defense work often goes unnoticed unless there's a problem. Keep at it. Despite what they say, I think we are starting to win.

This page intentionally left blank

About the Author

Kyle Rankin is a long-time systems administrator with a particular focus on infrastructure security, architecture, automation, and troubleshooting. In addition to this book he is the author of *DevOps Troubleshooting* (Addison-Wesley, 2012), *The Official Ubuntu Server Book, Third Edition* (Prentice Hall, 2013), and *Knoppix Hacks, Second Edition* (O'Reilly, 2007), among others. Rankin is an award-winning columnist for *Linux Journal* magazine, and he chairs the Purism advisory board. He speaks frequently on open-source software and security, including at O'Reilly Security Conference, CactusCon, SCALE, OSCON, LinuxWorld Expo, Penguicon, and a number of Linux Users' Groups.

This page intentionally left blank

3

Server Security

If someone is going to compromise your server, the most likely attack will either be through a vulnerability in a web application or other service the server hosts or through SSH. In other chapters, we cover hardening steps for common applications your server may host, so this chapter focuses more on general techniques to secure just about any server you have, whether it's hosting a website, email, DNS, or something completely different.

This chapter includes a number of different techniques to harden SSH and also covers how to limit the damage an attacker or even a malicious employee can do if she does get access to the server with tools like AppArmor and sudo. We also cover disk encryption to protect data at rest and how to set up a remote syslog server to make it more difficult for an attacker to cover her tracks.

Section 1: Server Security Fundamentals

Before we get into specific hardening techniques, we'll start with some fundamental server security practices. When attempting to secure your server, it's important to approach it with the appropriate mindset. With that in mind, there are a few principles you should apply no matter what your server does.

Fundamental Server Security Practices

Fundamental server security practices include the principle of least privilege, keeping it simple, and keeping your servers up to date.

The Principle of Least Privilege

We apply the principle of least privilege throughout the book (such as in Chapter 4, "Network," when we discuss firewall rules), but in general on a host you want users and applications to have only the privileges they need to do their job and nothing more. For instance, while all developers may have accounts on servers, you may restrict root access to systems administrators. You might also go further and only allow your average developer to have shell access to the development environment and restrict production server access only to systems administrators or other support staff that must have it.

When applied to applications, the principle of least privilege means that your application should only run as root if it is absolutely necessary; otherwise, it should run as some other account on the system. One common reason applications need root privileges is

to open a low port (all ports below 1025 require root privileges to open). Web servers provide a good example of this principle in that they must be root to open network ports 80 and 443; however, once those ports are open, the average worker process for the web server runs as a less-privileged user (such as the www-data user). It is common practice these days for web applications themselves to pick some other high port like 3000 or 8080 to listen on so that they can run as a normal user.

Keep It Simple

A simple server is easier to secure than a complicated one. Avoid installing and running extra services (especially network-facing services) that you don't need. That way, you have fewer applications that can have security holes and fewer applications to keep track of with security patches. It will be easier for your team and for external auditors to validate your configuration if you try to keep files and executables in their standard places and try to stick to default configurations when possible.

Simplicity is also important when designing the overall architecture of your environment. The more complicated your architecture and the more moving parts, the more difficult to understand and the more difficult to secure. If your network diagram looks like a bad plate of spaghetti, you may want to consider simplifying how servers communicate in the environment. Once we get to the network security chapter (Chapter 4) this will make that task simpler as well.

Keep Your Servers Up to Date

New vulnerabilities are found in applications or libraries all the time. One easy way to stay on top of security for your servers is to subscribe to the security mailing list for your distribution and then make sure, as security vulnerabilities are announced, that you prioritize patching servers. The more homogeneous your environment, the easier it will be to keep up with different versions of software, so you will have an easier time if you can stick to a single Linux distribution and a particular version of that distribution. Distribution security mailing lists won't cover any third-party software you run, however, so you will need to sign up for any security advisory lists those products provide.

SSH Configuration

One of the most common services on just about every server is SSH. While in the past administrators used tools like telnet that sent everything (including passwords!) over plain text, SSH encrypts the communications between you and your server. While that in and of itself is a security improvement, unfortunately it's not enough. In this section, we go over some basic SSH-hardening techniques you should be employing on all of your servers.

Disable Root Login

One of the simplest things you can do to make your SSH configuration more secure is to disable root logins. While later in this chapter we will talk about how you can avoid password logins to root via the sudo utility (and some systems default to that approach),

in this case we are talking about restricting the ability to log in as the root user whether via password, SSH keys, or any other method. Because of how much power the root user has, it's simply safer to remove the possibility that an attacker could directly log in with root privileges. Instead, have administrators log in as a regular user and then use local tools like sudo to become root.

To disable root login on a server, simply edit the SSH server configuration file (usually found at /etc/ssh/sshd_config config) and change

```
PermitRootLogin yes
```

to

```
PermitRootLogin no
```

and then restart the SSH service, which, depending on your system, may be one of the following:

```
$ sudo service ssh restart
$ sudo service sshd restart
```

Disable Protocol 1

The old SSH Protocol 1 has a number of known security vulnerabilities, so if your distribution doesn't already disable it you should do so. Locate the Protocol line in /etc/ssh/sshd_config and ensure it says

```
Protocol 2
```

If you did have to make a change to the file, be sure to restart the SSH service.

Sudo

In the old days when an administrator needed to do something as root, he either logged in directly as the root user or used a tool like su to become root. This approach had some problems, however. For one, it encouraged users to stay logged in as root. Because root can do pretty much anything you would want to do on the system, mistakes made as root could have much worse consequences than those made as a regular user. Also, from an administrative overhead standpoint, if you have multiple systems administrators who all have root access to a server, you would have to create a shared password that everyone knew. When an administrator inevitably left the company, the remaining team had to scramble to change the passwords for all of the shared accounts.

Sudo helps address many of these security concerns and provides a much stronger security model that helps you stick to the principle of least privilege. With sudo, an administrator can define groups of users who can perform tasks as other users including root. Sudo has a few specific advantages over su:

- Each user types his own password, not the password of the privileged account.

  This means you no longer have to manage shared passwords for privileged accounts. If a user leaves the company, you only have to disable her account. This also means

that administrators don't need to maintain passwords at all for role accounts (including root) on the system, so users or attackers can't get privileges they shouldn't have by guessing a password.

- Sudo allows fine-grained access control.

  With su, accessing a user's privileges is an all-or-nothing affair. If I can su to root, I can do anything I want as the root user. While you can certainly create sudo rules that allow the same level of access, you can also restrict users so that they can only run specific commands as root or as another user.

- Sudo makes it easier to stay out of privileged accounts.

  While you can certainly use sudo to get a complete root shell, the simplest invocations of sudo are just sudo followed by the command you want to run as root. This makes it easy to run privileged commands when you need to, and the rest of the time operate as your regular user.

- Sudo provides an audit trail.

  When a user on the system uses sudo, it looks at what user used sudo, what command was run, and when it was run. It also logs when a user tries to access sudo privileges they don't have. This provides a nice audit trail an administrator can use later to track down unauthorized access attempts on the system.

Sudo Examples and Best Practices

Sudo, like most access control systems, provides an extensive set of configuration options and methods to group users, roles, and commands. That configuration is usually found in /etc/sudoers although modern systems now often include an /etc/sudoers.d/ directory where one can better organize specific sets of sudo rules into his own files. The sudoers man page (type "man sudoers") goes into exhaustive detail on how to build your own complex sudo rules, and there are also plenty of other guides available. Instead of rehashing that documentation here, I describe some best practices when it comes to sudo rules and provide a few examples of useful sudo rules along the way. To get started, though, let's break down a generic sudo command:

```
root   ALL=(ALL) ALL
```

This command allows the root user to run any command as any user on any system. The first column is the user or group that the sudo rule applies to; in this case, the root user. The second column allows you to specify specific hosts this sudo rule applies to, or ALL if it applies on any host. The next entry in parentheses lists which user or users (separated by commas in the case of more than one user) can run commands—in this case, all users. The final column is a comma-separated list of specific executables on the system that you can run with these elevated privileges. In this case, it's all commands.

- Use visudo to edit /etc/sudoers.

  It may be tempting to just fire up your preferred text editor and edit /etc/sudoers directly, but the problem is that if you accidentally introduce a syntax error into /etc/sudoers you could lock yourself out of root access completely! When you use the

visudo tool, it performs a syntax validation on the file before it saves it, so you don't risk writing an invalid file.

- Grant access to groups, not specific users.

  This is more for ease of administration than specifically for security, but sudo allows you to grant access to a group on the system instead of a specific user. For instance, here are some examples of sudo rules you may have on your system to grant administrators root access:

  ```
  %admin  ALL=(ALL:ALL) ALL
  %wheel  ALL=(ALL:ALL) ALL
  %sudo   ALL=(ALL:ALL) ALL
  ```

  Each of these rules is an equivalent to root access. They let you become any user on the system and run any command you want as that user. The admin, wheel, and sudo groups are common groups on the system that a distribution might use to define who can become root.

  For a more useful example, let's say that you administer some tomcat servers and the developers need access to the local tomcat user in the development environment so they can troubleshoot their code. If we had all of their users in a group called developers, for instance, we could add the following rule to /etc/sudoers:

  ```
  %developers ALL=(tomcat) ALL
  ```

- Restrict access to specific commands as much as possible.

  While it's certainly easier to just allow someone to run all commands as a user, if we want to follow the principle of least privilege, we want to grant users access to only the privileged commands they need. This is particularly true when granting root access. For instance, if the database administrators (DBAs) needed access to run the psql command as postgres users so they could have more control over system-level database configuration, the lazy way would be to add a rule like the following:

  ```
  %dbas ALL=(postgres) ALL
  ```

  The problem is I don't necessarily want or need the DBAs to do more than run psql, so I could restrict the rule to just the command they need:

  ```
  %dbas ALL=(postgres) /usr/bin/psql
  ```

- Always use the full path to scripts.

  When writing sudo rules, always make sure to list the complete path to the executable you intend the user to run. Otherwise, if I had just listed psql instead of /usr/bin/psql, a malicious user could create a local script, name it psql, and have it do whatever she wanted.

- Write wrapper scripts to restrict risky commands to specific arguments.

  In many cases when you write sudo rules, you end up granting more powers than a user really needs. For instance, if I wanted to allow a user to restart the Nginx service, I could grant him access to the service command:

  ```
  bob ALL=(root) /usr/sbin/service
  ```

That would certainly give him the ability to restart Nginx, but he would also be able to start and stop any other service on the system. In this circumstance, it's better to create a small wrapper script named /usr/local/bin/restart_nginx like the following:

```
#!/bin/bash

/usr/sbin/service nginx restart
```

Then I would write a sudo rule that just allowed access to that script:

```
bob ALL=(root) /usr/local/bin/restart_nginx
```

If I wanted to allow bob to stop and start nginx as well, I could either modify the existing script to accept (and thoroughly validate) input, or I could create two new scripts along the same lines as the restart for the stop and start functions. In the latter case, I would update the sudo rule to be the following:

```
bob ALL=(root) /usr/local/bin/restart_nginx, /usr/local/bin/stop_nginx, /usr/
➥local/bin/start_nginx
```

Make sure that your wrapper scripts are only owned and writable only by root (chmod 775). In general, be careful about executing any scripts that a user can break out of and run shell commands from (such as vi).

- Resist writing NOPASSWD sudo rules unless absolutely necessary.

  Sudo provides a flag called NOPASSWD that doesn't require the user to enter a password when executing sudo. This can be a time saver; however, it removes one of the primary protections you have with sudo—namely, that a user has to authenticate herself to the system with her password before sudo allows her to run a command.

  That said, there are valid reasons to use the NOPASSWD flag, in particular if you want to execute a command from a role account on the system that may not have a password itself. For instance, you might want to allow the postgres database user to be able to trigger a cron job that runs a special database backup script as root, but the postgres role account doesn't have a password. In that case, you would add a sudo rule like the following:

  ```
  postgres ALL=(root) NOPASSWD: /usr/local/bin/backup_databases
  ```

Section 2: Intermediate Server-Hardening Techniques

Intermediate server-hardening techniques have to do with SSH key authentication, AppArmor, and remote logging.

SSH Key Authentication

Most administrators access their machines over SSH, and unfortunately, sometimes hackers do too! In fact, if you have a server exposed to the public Internet and have ever bothered to check your authentication logs (/var/log/auth.log on Debian-based systems),

you might have been surprised at just how many ssh attempts your machine constantly gets. What's happening here is called an SSH brute force attack. A number of attackers have realized that often the easiest way to compromise a Linux server is to guess a user's password. If one user (or common role account, like oracle or nagios for instance) happens to use a password that's in an attacker's dictionary, then it's just a matter of time before a script guesses it.

So how do you protect against an SSH brute force attack? One way would be to audit user passwords and enforce a strict password-complexity policy. Another might be to pick a different port for SSH, hoping that obscurity will save you. Yet another involves setting up systems that parse through SSH attempts and modify your firewall rules if too many attempts come from a single IP. Despite the fact that you can risk locking yourself out with systems like this, attackers have already moved on and will often only make a few attempts from a single IP in their vast botnet. Each of these methods can help reduce the risk of a successful SSH brute force attack, but it can't eliminate it completely.

If you want to eliminate SSH brute force attacks completely, the best way is also one of the simplest: eliminate password SSH logins. If you remove password SSH logins from the equation, then attackers can guess all of the passwords they want, and even if they guess right, SSH won't allow them to log in.

So if you remove password logins, how do you log in to SSH? The most common replacement for password-based login for SSH is to use SSH keypairs. With SSH keypairs, your client (a laptop or some other server) has both a public and private key. The private key is treated like a secret and stays on your personal machine, and the public key is copied to the ~/.ssh/authorized_keys file on the remote servers you want to log into.

Create SSH Keys

The first step is to create your SSH keypair. This is done via the ssh-keygen tool and while it accepts a large number of options and key types, we will use one that should work across a large number of servers:

```
$ ssh-keygen -t rsa -b 4096
```

The -t option selects the key type (RSA) and -b selects the bit size for the key (4096 bit), and this 4096-bit RSA key should be acceptable currently. When you run the command, it will prompt you for an optional passphrase to use to unlock the key. If you don't select a passphrase, then you can ssh into remote servers without having to type in a password. The downside is that if anyone gets access to your private key (by default in ~/.ssh/id_rsa), then he can immediately use it to ssh into your servers. I recommend setting a passphrase, and in a later section I talk about how to use ssh-agent to cache your passphrase for a period of time (much like sudo passwords are often cached for a few minutes so you don't have to type them with every command).

Once the command completes, you will have two new files: the private key at ~/.ssh/id_rsa and the public key at ~/.ssh/id_rsa.pub. The public key is safe to share with other people and it is the file that will get copied to remote servers, but the private key should be protected like your passwords or any other secrets and not shared with anyone.

You may want to consider creating different SSH keys for different purposes. Like using different passwords for different accounts, having different SSH keys for different accounts applies the principle of compartmentalization to your SSH keys and will help protect you if one key gets compromised. If you want to store a keypair with a different file name than the default, use the -f option to specify a different file. For instance, if you use the same computer for personal and work use, you will want to create a separate keypair for each environment:

```
$ ssh-keygen -t rsa -b 4096 -f ~/.ssh/workkey
```

The preceding command creates a workkey and workkey.pub file inside the ~/.ssh/ directory.

Copy SSH Keys to Other Hosts

Once you have your SSH keys in place, you will need to copy the contents of your public key to the ~/.ssh/authorized_keys file on the remote server. While you could just ssh into the remote server and do this by hand, a tool called ssh-copy-id has been provided to make this easy. For instance, if I wanted to copy my public key to a server called web1.example.com and my username was kyle, I would type:

```
$ ssh-copy-id kyle@web1.example.com
```

Replace the user and server with the username and server that you would use to log in to your remote server. At this point it will still prompt you for the password you use to log in to the remote machine, but once the command completes it will be the last time! Just like with regular SSH logins, if your local username is the same as the user on the remote server, you can omit it from the command. By default ssh-copy-id will copy your id_rsa.pub file, but if your keypair has a different name, then use the -i argument to specify a different public key. So if we wanted to use the custom workkey file we created previously, we would type:

```
$ ssh-copy-id -i ~/.ssh/workkey.pub kyle@web1.example.com
```

The nice thing about the ssh-copy-id command is that it takes care of setting the proper permissions on the ~/.ssh directory if it doesn't already exist (it should be owned by its own user, with 700 permissions), and it will also create the authorized_keys file if it needs to. This will help you avoid a lot of the headaches that come along with setting up SSH keys resulting from improper permissions on the local or remote ~/.ssh directory or local key files.

Once the ssh-copy-id command completes, you should be able to ssh into the remote server and not be prompted for the remote password. Now if you did set a passphrase for your SSH key, you will be prompted for that, but hopefully you chose a different passphrase for the key than the password you use on the remote server so it's easier to demonstrate that keys work.

Disable Password Authentication

Once you can ssh to the machine using keys, you are ready to disable password authentication. You will want to be careful with this step, of course, because if for some

reason your keys didn't work and you disable password authentication, you can risk locking yourself out of the server. If you are transitioning a machine being used by a number of people from password authentication to keys, you will want to make sure that everyone has pushed keys to the server before you proceed any further; otherwise, someone with root privileges will need to update their ~/.ssh/authorized_keys file with their public key by hand.

To disable password authentication, ssh into the remote server and get root privileges. Then edit the /etc/ssh/sshd_config file and change

```
PasswordAuthentication yes
```

to

```
PasswordAuthentication no
```

and then restart the SSH service which, depending on your system, may be one of the following:

```
$ sudo service ssh restart
$ sudo service sshd restart
```

Now, you don't want to risk locking yourself out, so keep your current SSH session active. Instead, open a new terminal and attempt to ssh into the server. If you can ssh in, then your key works and you are done. If not, run ssh with the -vvv option to get more verbose errors. To be safe, also undo the change to /etc/ssh/sshd_config and restart your SSH service to make sure you don't get completely locked out while you perform troubleshooting.

Working with Password-Protected SSH Keys

Some administrators enjoy the convenience of SSH keys that were created without a password. You can ssh to all your servers immediately without having to type in a password, and that can be pretty convenient. If you use a source control management tool like Git over ssh, you probably also want to avoid having to type in a password every time you push or pull from remote repositories. The downside to this approach is that without a password-protected SSH key, the security of your servers is only as good as the security behind your private SSH key. If someone gets access to your ~/.ssh/id_rsa file, they can immediately access any servers you can.

With a password-protected SSH key, even if your private key gets compromised, the attacker still needs to guess the password to unlock it. At the very least, that gives you time to create and deploy a new key, and depending on the attacker, the password-protected key may never be compromised. Password-protected keys are particularly important if you happen to store any keys on systems where you share root with other administrators. Without a password, any administrator with root on the system could log in to servers with your credentials.

You don't have to sacrifice convenience with a password-protected SSH key. There are tools in place that make it almost as convenient as any other method while giving you

a great deal of security. The main tool that makes SSH key passwords more manageable is the ssh-add utility. This tool is part of the ssh-agent utility; it allows you to type the password once and caches the unlocked key in RAM using SSH agent. Most Linux desktop systems these days have SSH agent running in the background (or via a wrapper like Gnome keyring). By default, it caches the key in RAM indefinitely (until the system powers down); however, I don't recommend that practice. Instead, I like to use ssh-add much like sudo password caching. I specify a particular time period to cache the key, after which I will be prompted for a password again.

For instance, if I wanted to cache the key for 15 minutes, much like sudo on some systems, I could type:

```
$ ssh-add -t 15m
Enter passphrase for /home/kyle/.ssh/id_rsa:
Identity added: /home/kyle/.ssh/id_rsa (/home/kyle/.ssh/id_rsa)
Lifetime set to 900 seconds
```

Note that I was able to specify the time in minutes to the -t argument by appending "m" to the end; otherwise, it assumes the number is in seconds. You will probably want to cache the key for a bit longer than that, though; for instance, to cache the key for an hour you could type:

```
$ ssh-add -t 1h
Enter passphrase for /home/kyle/.ssh/id_rsa:
Identity added: /home/kyle/.ssh/id_rsa (/home/kyle/.ssh/id_rsa)
Lifetime set to 3600 seconds
```

From now until the key expires, you can ssh into servers and use tools like Git without being prompted for the password. Once your time is up, the next time you use SSH you will be prompted for a password and can choose to run ssh-add again. If the key you want to add is not in the default location, just add the path to the key at the end of the ssh-add command:

```
$ ssh-add -t 1h ~/.ssh/workkey
```

What I like to do is use this tool like a personal timer. When I start work in the morning, I calculate the number of hours or minutes from now until when I want to go to lunch and set the ssh-add timer to that number. Then I work as usual and once the next Git push or ssh command prompts me for a password, I realize it's time to go grab lunch. When I get back from lunch I do the same thing to notify me when it's time to leave for the day.

Of course, using a tool like this does mean that if an attacker were able to compromise your machine during one of these windows where the key is cached, she would be able to have all of the access you do without typing in a password. If you are working in an environment where that's too much of a risk, just keep your ssh-add times short, or run ssh-add -D to delete any cached keys any time you leave your computer. You could even potentially have your lock command or screensaver call this command so it happens every time you lock your computer.

AppArmor[1]

The UNIX permissions model has long been used to lock down access to users and programs. Even though it works well, there are still areas where extra access control can come in handy. For instance, many services still run as the root user, and therefore if they are exploited, the attacker potentially can run commands throughout the rest of the system as the root user. There are a number of ways to combat this problem, including sandboxes, chroot jails, and so on, but Ubuntu has included a system called AppArmor, installed by default, that adds access control to specific system services.

AppArmor is based on the security principle of least privilege; that is, it attempts to restrict programs to the minimal set of permissions they need to function. It works through a series of rules assigned to particular programs. These rules define, for instance, which files or directories a program is allowed to read and write to or only read from. When an application that is being managed by AppArmor violates these access controls, AppArmor steps in and prevents it and logs the event. A number of services include AppArmor profiles that are enforced by default, and more are being added in each Ubuntu release. In addition to the default profiles, the universe repository has an apparmor-profiles package you can install to add more profiles for other services. Once you learn the syntax for AppArmor rules, you can even add your own profiles.

Probably the simplest way to see how AppArmor works is to use an example program. The BIND DNS server is one program that is automatically managed by AppArmor under Ubuntu, so first I install the BIND package with sudo apt-get install bind9. Once the package is installed, I can use the aa-status program to see that AppArmor is already managing it:

```
$ sudo aa-status
apparmor module is loaded.
5 profiles are loaded.
5 profiles are in enforce mode.
/sbin/dhclient3
/usr/lib/NetworkManager/nm-dhcp-client.action
/usr/lib/connman/scripts/dhclient-script
/usr/sbin/named
/usr/sbin/tcpdump
0 profiles are in complain mode.
2 processes have profiles defined.
1 processes are in enforce mode :
/usr/sbin/named (5020)
0 processes are in complain mode.
1 processes are unconfined but have a profile defined.
/sbin/dhclient3 (607)
```

Here you can see that the /usr/sbin/named profile is loaded and in enforce mode, and that my currently running /usr/sbin/named process (PID 5020) is being managed by AppArmor.

---

1. Rankin, Kyle; Hill, Benjamain Mako, *The Official Ubuntu Server Book, Third Edition,* © 2014. Reprinted by permission of Pearson Education, Inc., New York, New York.

AppArmor Profiles

The AppArmor profiles are stored within /etc/apparmor.d/ and are named after the binary they manage. For instance, the profile for /usr/sbin/named is located at /etc/apparmor.d/usr.sbin.named. If you look at the contents of the file, you can get an idea of how AppArmor profiles work and what sort of protection they provide:

```
# vim:syntax=apparmor
# Last Modified: Fri Jun 1 16:43:22 2007
#include <tunables/global>
/usr/sbin/named {
  #include <abstractions/base>
  #include <abstractions/nameservice>

  capability net_bind_service,
  capability setgid,
  capability setuid,
  capability sys_chroot,

  # /etc/bind should be read-only for bind
  # /var/lib/bind is for dynamically updated zone (and journal) files.
  # /var/cache/bind is for slave/stub data, since we're not the origin
  #of it.
  # See /usr/share/doc/bind9/README.Debian.gz
  /etc/bind/** r,
  /var/lib/bind/** rw,
  /var/lib/bind/ rw,
  /var/cache/bind/** rw,
  /var/cache/bind/ rw,

  # some people like to put logs in /var/log/named/
  /var/log/named/** rw,

  # dnscvsutil package
  /var/lib/dnscvsutil/compiled/** rw,

  /proc/net/if_inet6 r,
  /usr/sbin/named mr,
  /var/run/bind/run/named.pid w,
  # support for resolvconf
  /var/run/bind/named.options r,
}
```

For instance, take a look at the following excerpt from that file:

```
/etc/bind/** r,
/var/lib/bind/** rw,
/var/lib/bind/ rw,
/var/cache/bind/** rw,
/var/cache/bind/ rw,
```

The syntax is pretty straightforward for these files. First there is a file or directory path, followed by the permissions that are allowed. Globs are also allowed, so, for instance, /etc/bind/**** applies to all the files below the /etc/bind directory recursively. A single ***** would apply only to files within the current directory. In the case of that rule, you can see that /usr/sbin/named is allowed only to read files in that directory and not write there.

This makes sense, since that directory contains only BIND configuration files—the named program should never need to write there. The second line in the excerpt allows named to read and write to files or directories under /var/lib/bind/. This also makes sense because BIND might (among other things) store slave zone files here, and since those files are written to every time the zone changes, named needs permission to write there.

Enforce and Complain Modes

You might have noticed that the aa-status output mentions two modes: enforce and complain. In enforce mode, AppArmor actively blocks any attempts by a program to violate its profile. In complain mode, AppArmor simply logs the attempt but allows it to happen. The aa-enforce and aa-complain programs allow you to change a profile to be in enforce or complain mode, respectively. So if my /usr/sbin/named program did need to write to a file in /etc/bind or some other directory that wasn't allowed, I could either modify the AppArmor profile to allow it or I could set it to complain mode:

```
$ sudo aa-complain /usr/sbin/named
Setting /usr/sbin/named to complain mode
```

If later I decided that I wanted the rule to be enforced again, I would use the aa-enforce command in the same way:

```
$ sudo aa-enforce /usr/sbin/named
Setting /usr/sbin/named to enforce mode
```

If I had decided to modify the default rule set at /etc/apparmor.d/usr.sbin.named, I would need to be sure to reload AppArmor so it would see the changes. You can run AppArmor's init script and pass it the reload option to accomplish this:

```
$ sudo /etc/init.d/apparmor reload
```

Be careful when you modify AppArmor rules. When you first start to modify rules, you might want to set that particular rule into complain mode and then monitor /var/log/syslog for any violations. For instance, if /usr/sbin/named were in enforce mode and I had commented out the line in the /usr/sbin/named profile that granted read access to /etc/bind/**, then reloaded AppArmor and restarted BIND, not only would BIND not start (since it couldn't read its config files), I would get a nice log entry in /var/log/syslog from the kernel to report the denied attempt:

```
Jan 7 19:03:02 kickseed kernel: [ 2311.120236]
  audit(1231383782.081:3): type=1503 operation="inode_permission"
  requested_mask="::r" denied_mask="::r" name="/etc/bind/named.conf"
  pid=5225 profile="/usr/sbin/named" namespace="default"
```

Ubuntu AppArmor Conventions

The following list details the common directories and files AppArmor uses, including where it stores configuration files and where it logs:

- **/etc/apparmor/**: This directory contains the main configuration files for the AppArmor program, but note that it does *not* contain AppArmor rules.

- **/etc/apparmor.d/**: You will find all the AppArmor rules under this directory along with subdirectories that contain different sets of include files to which certain rule sets refer.
- **/etc/init.d/apparmor**: This is the AppArmor init script. By default, AppArmor is enabled.
- **/var/log/apparmor/**: AppArmor stores its logs under this directory.
- **/var/log/syslog**: When an AppArmor rule is violated in either enforce or complain mode, the kernel generates a log entry under the standard system log.

Remote Logging

Logs are an important troubleshooting tool on a server but they are particularly useful after an attacker has compromised a server. System logs show every local and SSH login attempt, any attempt to use sudo, kernel modules that are loaded, extra file systems that are mounted, and if you use a software firewall with logging enabled it might show interesting networking traffic from the attacker. On web, database, or application servers, you also get extra logging from attempting accesses of those systems.

The problem is that attackers know how useful and revealing logs are, too, so any reasonably intelligent attacker is going to try to modify any log on the system that might show her tracks. Also, one of the first things many rootkits and other attack scripts do is wipe the local logs and make sure their scripts don't generate new logs.

As a security-conscious administrator, you will find it important that all logs on a system that might be useful after an attack also be stored on a separate system. Centralized logging is useful for overall troubleshooting, but it also makes it that much more difficult for an attacker to cover her tracks since it would mean not only compromising the initial server but also finding a way to compromise your remote logging server. Depending on your company, you may also have regulatory requirements to log certain critical logs (like login attempts) to a separate server for longer-term storage.

There are a number of systems available such as Splunk and Logstash, among others, that not only collect logs from servers but can also index the logs and provide an interface an administrator can use to search through the logs quickly. Many of these services provide their own agent that can be installed on the system to ease collection of logs; however, just about all of them also support log collection via standard syslog network protocol.

Instead of going through all the logging software out there, in this section I describe how to configure a client to ship logs to a remote syslog server, and in case you don't have a central syslog server in place yet, I follow up with a few simple steps to configure a basic centralized syslog server. I've chosen rsyslog as my example syslog server because it supports classic syslog configuration syntax, has a number of extra features for administrators that want to fine-tune the server, and should be available for all major Linux distributions.

Client-Side Remote Syslog Configuration

It is relatively straightforward to configure a syslog client to ship logs to a remote server. Essentially you can go into your syslog configuration (in the case of rsyslog, this

is at /etc/rsyslog.conf in many cases along with independent configuration files in /etc/rsyslog.d) and find the configuration for the log file that you want to ship remotely. For instance, I may have a regulatory requirement that all authentication logs be shipped to a remote source. On a Debian-based system, those logs are in /var/log/auth.log, and if I look through my configuration files I should see the line that describes what type of events show up in this log:

```
auth,authpriv.*     /var/log/auth.log
```

Since I want to ship these logs remotely as well, I need to add a new line almost identical to the preceding line, except I would replace the path to the local log file with the location of the remote syslog server. For instance, if I named the remote syslog server "syslog1.example.com," I would either add a line below the preceding line or create a new configuration file under /etc/rsyslog.d with the following syntax:

```
auth,authpriv.*     @syslog1.example.com:514
```

The syntax for this line is an @ sign for User Datagram Protocol (UDP), or @@ for Transmission Control Protocol (TCP), the hostname or IP address to ship the logs to, and optionally a colon and the port to use. If you don't specify a port, it will use the default syslog port at 514. Now restart the rsyslog service to use the new configuration:

```
$ sudo service rsyslog restart
```

In the preceding example, I use UDP to ship my logs. In the past, UDP was preferred since it saved on overall network traffic when shipping logs from a large number of servers; however, with UDP you do risk losing logs if the network gets congested. An attacker could even attempt to congest the network to prevent logs from getting to a remote log server. While TCP does provide extra overhead, the assurance that your logs will not get dropped is worth the extra overhead. So, I would change the previous configuration line to

```
auth,authpriv.*     @@syslog1.example.com:514
```

If you find after some time that this does create too much network load, you can always revert back to UDP.

Server-Side Remote Syslog Configuration

If you don't already have some sort of central logging server in place, it's relatively simple to create one with rsyslog. Once rsyslog is installed, you will want to make sure that remote servers are allowed to connect to port 514 UDP and TCP on this system, so make any necessary firewall adjustments. Next, add the following options to your rsyslog configuration file (either /etc/rsyslog.conf directly, or by adding an extra file under /etc/rsyslog.d):

```
$ModLoad imudp
$UDPServerRun 514
$ModLoad imtcp
$InputTCPServerRun 514
```

This will tell rsyslog to listen on port 514 both for UDP and TCP. You also will want to restrict what IPs can communicate with your rsyslog server, so add extra lines that restrict which networks can send logs to this server:

```
$AllowedSender UDP, 192.168.0.0/16, 10.0.0.0/8, 54.12.12.1
$AllowedSender TCP, 192.168.0.0/16, 10.0.0.0/8, 54.12.12.1
```

These lines allow access from the internal 192.168.x.x and 10.x.x.x networks as well as an external server at 54.12.12.1. Obviously, you will want to change the IPs mentioned here to reflect your network.

If you were to restart rsyslog at this point, the local system logs would grow not just with logs from the local host, but also with any logs from remote systems. This can make it difficult to parse through and find logs just for a specific host, so we also want to tell rsyslog to organize logs in directories based on hostname. This requires that we define a template for each type of log file that we want to create. In our client example, we showed how to ship auth.log logs to a remote server, so here we follow up with an example configuration that will accept those logs and store them locally with a custom directory for each host based on its hostname:

```
$template Rauth,"/var/log/%HOSTNAME%/auth.log"
auth.*,authpriv.*  ?Rauth`
```

In the first line, I define a new template I named Rauth and then specify where to store logs for that template. The second line looks much like the configuration we used on our client, only in this case at the end of the line I put a question mark and the name of my custom template. Once your configuration is in place, you can restart rsyslog with:

```
$ sudo service rsyslog restart
```

You should start to see directories being created under /var/log for each host that sends the server authentication logs. You can repeat the preceding template lines for each of the log types you want to support; just remember that you need to define the template before you can use it.

Section 3: Advanced Server-Hardening Techniques

Depending on your level of threat, you may want to add some additional hardening techniques to each of your servers. The advanced server-hardening techniques we cover include server disk encryption, secure NTP alternatives, and two-factor authentication with SSH.

Server Disk Encryption

Like many more advanced security-hardening techniques, disk encryption is one of those security practices that many administrators skip unless they are required to engage it by regulations, by the sensitivity of the data they store, or by the presence of an overall high-security environment. After all, it requires extra work to set up, it can lower your overall disk performance, and it can require that you manually intervene to enter a passphrase to unlock a disk whenever you boot. As you consider whether you should encrypt your disks, it is important to recognize what security it provides and what security it can't provide.

- Encryption protects data at rest. Disk encryption will encrypt data as it is written to disk but provides the data in unencrypted form while the file system is mounted. When the disk is unmounted (or the server is powered off), the data is encrypted and can't be read unless you know the passphrase.
- Encryption does *not* protect the live file system. If an attacker compromises a server while the encrypted disk is mounted (which would usually be the case for most running servers), he will be able to read the data as though it were any other unencrypted file system. In addition, if the attacker has root privileges, he can also retrieve the decryption key from RAM.
- The encryption is only as strong as your passphrase. If you pick a weak password for your disk encryption, then an attacker will eventually guess it.

Root Disk Encryption

The examples I give next are to encrypt non-root disks. For servers, it's simpler to segregate your sensitive data to an encrypted disk and leave the OS root partition unencrypted. That way you could set up your system to be able to boot to a command prompt and be accessible over the network in the event of a reboot without prompting you for a passphrase. That said, if your environment is sensitive enough that even the root disk must be encrypted, the simplest way to set it up is via your Linux distribution's installer, either manually in the partitioning section of your installation disk, or through an automated installation tool like kickstart or preseed.

Non-root Disk Encryption

I'm assuming that if you chose to encrypt your root disk, you probably encrypted all the remaining disks on your server during installation. However, if you haven't chosen to encrypt everything, you likely have a disk or partition you intend to use for sensitive information. In the examples that follow, we use the Linux Unified Key Setup (LUKS) disk encryption tools and, in particular, we use the cryptsetup script that simplifies the process of creating a new LUKS volume. If cryptsetup isn't already installed on your server, the package of the same name should be available for your distribution.

In the following example, we will set up an encrypted volume on the /dev/sdb disk but you could also select a partition on the disk instead. All commands will require root permissions. In the end, we will have a disk device at /dev/mapper/crypt1 we can format, mount, and treat like any other disk.

The first step is to use the cryptsetup tool to create the initial encrypted drive with your chosen passphrase and format it with random data before you use it:

```
$ sudo cryptsetup --verbose --verify-passphrase luksFormat /dev/sdb
WARNING!
========
This will overwrite data on /dev/sdb irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase:
Verify passphrase:
Command successful.
```

At this point, you have a LUKS encrypted disk on /dev/sdb, but before you can use it you need to open the device (which will prompt you for the passphrase) and map it to a device under /dev/mapper/ that you can mount:

```
$ sudo cryptsetup luksOpen /dev/sdb crypt1
Enter passphrase for /dev/sdb:
```

The syntax for this command is to pass cryptsetup the luksOpen command followed by the LUKS device you want to access, and finally the label you want to assign to this device. The label will be the name that shows up under /dev/mapper, so in the preceding example, after the command completes, I will have a device under /dev/mapper/crypt1.

Once /dev/mapper/crypt1 exists, I can format it with a file system and mount it like any other drive:

```
$ sudo mkfs -t ext4 /dev/mapper/crypt1
$ sudo mount /dev/mapper/crypt1 /mnt
```

You will likely want to set this up so that the device shows up in the same way after every boot. Like with the /etc/fstab file you use to map devices to mount point at boot, there is an /etc/crypttab file you can use to map a particular device to the label you want to assign to it. Like with modern /etc/fstab files, it's recommended that you reference the UUID assigned to the device. Use the blkid utility to retrieve the UUID:

```
$ sudo blkid /dev/sdb
/dev/sdb: UUID="0456899f-429f-43c7-a6e3-bb577458f92e" TYPE="crypto_LUKS"
```

Then update /etc/cryptab by specifying the label you want to assign the volume (crypt1 in our example), the full path to the disk, then "none" for the key file field, and then "luks" as the final option. The result in our case would look like this:

```
$ cat /etc/crypttab
# <target name> <source device>         <key file>      <options>
crypt1 /dev/disk/by-uuid/0456899f-429f-43c7-a6e3-bb577458f92e none luks
```

If you do set up /etc/crypttab, you will be prompted at boot for a passphrase. Note in our example we did not set up a key file. This was intentional because a key file on the unencrypted root file system would probably be available to an attacker who had access to the powered-off server, and then she would be able to decrypt the disk.

Secure NTP Alternatives

Accurate time is important on servers, not just as a way to synchronize log output between hosts, but because most clustering software relies on cluster members having an accurate clock. Most hosts use a service called Network Time Protocol (NTP) to query a remote NTP server for accurate time. You need root permissions to set the time on a server, so typically the NTP daemon (ntpd) ends up running in the background on your system as root.

I would imagine most administrators don't think about NTP when they think about security. It is one of those protocols you take for granted; however, most administrators ultimately rely on an external accurate time source (like nist.gov) for NTP. Because

NTP uses the UDP protocol, it might be possible for an attacker to send a malicious, spoofed NTP reply before the legitimate server. This reply could simply send the server an incorrect time, which could cause instability, or given that ntpd runs as root, if it didn't validate the NTP reply in a secure way, there is potential for a man-in-the-middle attacker to send a malicious reply that could execute code as root.

One alternative to NTP is tlsdate, an open-source project that takes advantage of the fact that the TLS handshake contains time information. With tlsdate, you can start a TLS connection over TCP with a remote server that you trust and pull down its time. While the timestamps in TLS are not as precise as with NTP, they should be accurate enough for normal use. Since tlsdate uses TCP and uses TLS to validate the remote server, it is much more difficult for an attacker to send malicious replies back.

The tlsdate project is hosted at https://github.com/ioerror/tlsdate, and the general-purpose installation instructions can be found at https://github.com/ioerror/tlsdate/blob/master/INSTALL. That said, tlsdate is already packaged for a number of popular Linux distributions, so first use your standard package tool to search for the tlsdate package. If it doesn't exist, you can always download the source code from the aforementioned site and perform the standard compilation process:

```
./autogen.sh
./configure
make
make install
```

The tlsdate project includes a systemd or init script (depending on your distribution) that you can start with service tlsdated start. Once running, the script will notice network changes and will periodically keep the clock in sync in the background. If you want to test tlsdate manually, you can set the clock with the following command:

```
$ sudo tlsdate -V
Sat Jul 11 10:45:37 PDT 2015
```

By default, tlsdate uses google.com as a trusted server. On the command line, you can use the -H option to specify a different host:

```
$ sudo tlsdate -V -H myserver.com
```

If you want to change the default, edit /etc/tlsdate/tlsdated.conf and locate the source section:

```
# Host configuration.
source
        host google.com
        port 443
        proxy none
end
```

Change the host to whichever host you would like to use. For instance, you may want to pick a couple of hosts in your network that poll an external source for time and have the rest of your servers use those internal trusted hosts for time. Those internal trusted servers simply need to serve some kind of TLS service (such as HTTPS).

Two-Factor Authentication with SSH

Disabling password authentication on SSH and relying strictly on keys is a great first step to hardening SSH, but it still is not without its risks. First, while you may follow my advice and protect your SSH keys with a password, you can't guarantee that every user on the system does the same. This means if an attacker were able to access a computer for a short time, he could copy and use the keys to log in to your system. If you want to protect against this kind of attack, one approach is to require two-factor authentication for SSH.

With two-factor authentication, a user must provide both an SSH key and a separate token to log into the server. Time-based tokens are the most common, and in the past they required you to carry around an expensive device on your keychain that would update itself with a new token every 30 seconds. These days, there are a number of two-factor authentication solutions that work in software and can use your cell phone instead of a hardware token.

There are several different phone-based two-factor authentication libraries out there for ssh. Some work by the SSH client ForceCommand config option, some with a system-wide pluggable authentication modules (PAM) module. Some approaches are time-based, so they work even if your device is disconnected from a network, while others use SMS or phone calls to transfer the code. For this section, I've chosen the Google Authenticator library for a couple of reasons:

- It's been around for a number of years and is already packaged for a number of Linux distributions.
- The Google Authenticator client is available for multiple phone platforms.
- It uses PAM, so you can easily enable it system-wide without having to edit ssh config files for each user.
- It provides the user with backup codes she can write down in case her phone is ever stolen.

Install Google Authenticator

The Google Authenticator library is packaged for different platforms so, for instance, on Debian-based systems you can install it with

```
$ sudo apt-get install libpam-google-authenticator
```

If it isn't already packaged for your distribution, go to https://github.com/google/google-authenticator/ and follow the directions to download the software, build, and install it.

Configure User Accounts

Before we make PAM or SSH changes that might lock you out, you will want to at least configure Google Authenticator for your administrators. First, install the Google Authenticator application on your smartphone. It should be available via the same methods you use to install other applications.

Once the application is installed, the next step is to create a new Google Authenticator account on your phone from your server. To do this, log in to your account as your user

and then run google-authenticator. It will walk you through a series of questions, and it's safe to answer yes to every question, although since you should have already set up tlsdate on your server so its time is accurate, I recommend sticking with the default 90-second window instead of increasing it to 4 minutes. The output looks something like this:

```
kyle@debian:~$ google-authenticator

Do you want authentication tokens to be time-based (y/n) y
[URL for TOTP goes here]
[QR code goes here]
Your new secret key is: NONIJIZMPDJJC9VM
Your verification code is 781502
Your emergency scratch codes are:
  60140990
  16195496
  49259747
  24264864
  37385449

Do you want me to update your "/home/kyle/.google_authenticator" file (y/n) y

Do you want to disallow multiple uses of the same authentication
token? This restricts you to one login about every 30s, but it increases
your chances to notice or even prevent man-in-the-middle attacks (y/n) y

By default, tokens are good for 30 seconds and in order to compensate for
possible time-skew between the client and the server, we allow an extra
token before and after the current time. If you experience problems with poor
time synchronization, you can increase the window from its default
size of 1:30min to about 4min. Do you want to do so (y/n) n

If the computer that you are logging into isn't hardened against brute-force
login attempts, you can enable rate-limiting for the authentication module.
By default, this limits attackers to no more than 3 login attempts every 30s.
Do you want to enable rate-limiting (y/n) y
```

If you have libqrencode installed, this application will not only output a URL you can visit to add this account to your phone, it will also display a QR code on the console (I removed it from the preceding output). You can either scan that QR code with your phone, or enter the secret key that follows "Your new secret key is:" in the output.

The emergency scratch codes are one-time use codes that you can use if you ever lose or wipe your phone. Write them down and store them in a secure location apart from your phone.

Configure PAM and SSH

Once you have configured one or more administrators on the server, the next step is to configure PAM and SSH to use Google Authenticator. Open your SSH PAM configuration file (often at /etc/pam.d/sshd) and at the top of the file add

```
auth required pam_google_authenticator.so
```

On my systems, I noticed that once I enabled Google Authenticator and ChallengeResponseAuthentication in my SSH configuration file, logins would also

prompt me for a password after I entered my two-factor authentication code. I was able to disable this by commenting out

```
@include common-auth
```

in /etc/pam.d/sshd, although if you aren't on a Debian-based system, your PAM configuration may be a bit different.

Once the PAM file was updated, the final step was to update my SSH settings. Open /etc/ssh/sshd_config and locate the ChallengeResponseAuthentication setting. Make sure it's set to yes, or if it's not, in your sshd_config file add it:

```
ChallengeResponseAuthentication yes
```

Also, since we have disabled password authentication before, and are using key-based authentication, we will need to add an additional setting to the file, otherwise SSH will accept our key and never prompt us for our two-factor authentication code. Add the following line to the config file, as well:

```
AuthenticationMethods publickey,keyboard-interactive
```

Now you can restart ssh with one of the following commands:

```
$ sudo service ssh restart
$ sudo service sshd restart
```

Once SSH is restarted, the next time you log in you should get an additional prompt to enter the two-factor authentication code from your Google Authenticator app:

```
$ ssh kyle@web1.example.com
Authenticated with partial success.
Verification code:
```

From this point on, you will need to provide your two-factor token each time you log in.

Summary

No matter what service your server runs, there are certain basic hardening techniques you should apply. In this chapter, we focused specifically on hardening steps that apply to any server. In particular, we discussed hardening superuser access with sudo and the importance of remote logging. Also, given that almost every server these days uses SSH for remote administration, we covered a number of techniques to harden that service, from general hardening of SSH server configurations like disabling root logins to the use of SSH keys instead of password authentication. Finally, we discussed some advanced server-hardening techniques including adding two-factor authentication to SSH logins, server disk encryption, and alternatives to NTP.

Index

T

U

V