Apache Security
By Ivan Ristic

Publisher: O'Reilly
Pub Date: March 2005
ISBN: 0-596-00724-8
Pages: 420

This all-purpose guide for locking down Apache arms readers with all the information they need to securely deploy applications. Administrators and programmers alike will benefit from a concise introduction to the theory of securing Apache, plus a wealth of practical advice and real-life examples. Topics covered include installation, server sharing, logging and monitoring, web applications, PHP and SSL/TLS, and more.

Apache Security
By Ivan Ristic

Publisher: O'Reilly
Pub Date: March 2005
ISBN: 0-596-00724-8
Pages: 420

- Table of Contents
- Index
- Reviews
- Reader Reviews
- Errata
- Academic

Dedication

To my dear wife Jelena, who makes my life worth living.

Printed in the United States of America.

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

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

Preface

There is something about books that makes them one of the most precious things in the world. I've always admired people who write them, and I have always wanted to write one myself. The book you are now holding is a result of many years of work with the referenced Internet technologies and almost a year of hard work putting the words on paper. The preface may be the first thing you are reading, but it is the last thing I am writing. And I can tell you it has been quite a ride.

Aside from my great wish to be a writer in the first place, which only helped me in my effort to make the book as good as possible, there is a valid reason for its existence: a book of this profile is greatly needed by all those who are involved with web security. I, and many of the people I know, need it. I've come to depend on it in my day-to-day work, even though at the time of this writing it is not yet published. The reason this book is needed is that web security is affected by some diverse factors, which interact with each other in web systems and affect their security in varied, often subtle ways. Ultimately, what I tried to do was create one book to contain all the information one needs to secure an Apache-based system. My goal was to write a book I could safely recommend to anyone who is about to deploy on Apache, so I would be confident they would succeed provided they followed the advice in the book. You have, in your hands, the result of that effort.

Audience

This book aims to be a comprehensive Apache security resource. As such, it contains a lot of content on the intermediate and advanced levels. If you have previous experience with Apache, I expect you will have no trouble jumping to any part of the book straight away. If you are completely new to Apache, you will probably need to spend a little time learning the basics first, perhaps reading an Apache administration book or taking one of the many tutorials available online. Since Apache Security covers many diverse topics, it's likely that no matter what level of experience you have you are likely to have a solid starting point.

This book does not assume previous knowledge of security. Security concepts relevant for discussion are introduced and described wherever necessary. This is especially true for web application security, which has its own chapter.

The main thing you should need to do your job in addition to this book, is the Apache web server's excellent reference documentation (http://httpd.apache.org/docs/).

The book should be especially useful for the following groups:

System administrators

Their job is to make web systems secure. This book presents detailed guidance that enables system administrators to make informed decisions about which measures to take to enhance security.

Programmers

They need to understand how the environment in which their applications are deployed works. In addition, this book shows how certain programming errors lead to vulnerabilities and tells what to do to avoid such problems.

System architects

They need to know what system administrators and programmers do, and also need to understand how system design decisions affect overall security.

Web security professionals

They need to understand how the Apache platform works in order to assess the security of systems deployed on it.

Scope

At the time of this writing, two major Apache branches are widely used. The Apache 1.x branch is the well-known, and well-tested, web server that led Apache to dominate the web server market. The 2.0.x branch is the next-generation web server, but one that has suffered from the success of the previous branch. Apache 1 is so good that many of its users do not intend to upgrade in the near future. A third branch, 2.2.x will eventually become publicly available. Although no one can officially retire an older version, the new 2.2.x branch is a likely candidate for a version to replace Apache 1.3.x. The Apache branches have few configuration differences. If you are not a programmer (meaning you do not develop modules to extend Apache), a change from an older branch to a newer branch should be straightforward.

This book covers both current Apache branches. Wherever there are differences in the configuration for the two branches, such differences are explained. The 2.2.x branch is configured in practically the same way as the 2.0.x branch, so when the new branch goes officially public, the book will apply to it equally well.

Many web security issues are directly related to the operating system Apache runs on. For most of this book, your operating system is irrelevant. The advice I give applies no matter whether you are running some Unix flavor, Windows, or some other operating system. However, in most cases I will assume you are running Apache on a Unix platform. Though Apache runs well on Windows, Unix platforms offer another layer of configuration options and security features that make them a better choice for security-conscious deployments. Where examples related to the operating system are given, they are typically shown for Linux. But such examples are in general very easy to translate to other Unix platforms and, if you are running a different Unix platform, I trust you will have no problems with translation.

Contents of This Book

While doing research for the book, I discovered there are two types of people: those who read books from cover to cover and those who only read those parts that are of immediate interest. The book's structure (12 chapters and 1 appendix) aims to satisfy both camps. When read sequentially, the book examines how a secure system is built from the ground up, adding layer upon layer of security. However, since every chapter was written to cover a single topic in its entirety, you can read a few selected chapters and leave the rest for later. Make sure to read the first chapter, though, as it establishes the foundation for everything else.

Chapter 1, presents essential security principles, security terms, and a view of security as a continuous process. It goes on to discuss threat modeling, a technique used to analyze potential threats and establish defenses. The chapter ends with a discussion of three ways of looking at a web system (the user view, the network view, and the Apache view), each designed to emphasize a different security aspect. This chapter is dedicated to the strategy of deploying a system that is created to be secure and that is kept secure throughout its lifetime.

Chapter 2, gives comprehensive and detailed coverage of the Apache installation and configuration process, where the main goal is not to get up and running as quickly as possible but to create a secure installation on the first try. Various hardening techniques are presented along with discussions of the advantages and disadvantages of each.

Chapter 3, discusses PHP installation and configuration, following the same style established in Chapter 2. It begins with a discussion of and installation guidance for common PHP deployment models (as an Apache module or as a CGI), continues with descriptions of security-relevant configuration options (such as the safe mode), and concludes with advanced hardening techniques.

Chapter 4, discusses cryptography on a level sufficient for the reader to make informed decisions about it. The chapter first establishes the reasons cryptography is needed, then introduces SSL and discusses its strengths and weaknesses. Practical applications of SSL for Apache are covered through descriptions and examples of the use of *mod_ssl* and OpenSSL. This chapter also specifies the procedures for functioning as a certificate authority, which is required for high security installations.

Chapter 5, discusses some dangers of establishing a public presence on the Internet. A denial of service attack is, arguably, one of the worst problems you can experience. The problems discussed here include network attacks, configuration and programming issues that can make you harm your own system, local (internal) attacks, weaknesses of the Apache processing model, and traffic spikes. This chapter describes what can happen, and the actions you can take, before such attacks occur, to make your system more secure and reduce the potential effects of such attacks. It also gives guidance regarding what to do if such attacks still occur in spite of your efforts.

Chapter 6, discusses the problems that arise when common server resources must be shared with people you may not trust. Resource sharing usually leads to giving other people partial control of the web server. I present several ways to give partial control without giving too much. The practical problems this chapter aims to solve are shared hosting, working with developers, and hosting in environments with large numbers of system users (e.g., students).

Chapter 7, discusses the theory and practice of user identification, authentication (verifying a user is allowed to access the system), and authorization (verifying a user is allowed to access a particular resource). For Apache, this means coverage of HTTP-defined authentication protocols (Basic and Digest authentication), form-based and certificate-based authentication, and network-level access control. The last part of the chapter discusses single sign-on, where people can log in once and have access to several different resources.

Chapter 8, describes various ways Apache can be configured to extract interesting and relevant pieces of information, and record them for later analysis. Specialized logging modules, such as the ones that help detect problems that cause the server to crash, are also covered. The chapter then addresses log collection, centralization, and analysis. The end of the chapter covers operation monitoring, through log analysis in batch or real-time. A complete example of using *mod_status* and RRDtool to monitor Apache is presented.

Chapter 9, discusses a variety of security issues related to the environment in which the Apache web server exists. This chapters touches upon network security issues and gives references to web sites and books in which the subject

Online Companion

A book about technology cannot be complete without a companion web site. To fully appreciate this book, you need to visit http://www.apachesecurity.net, where I am making the relevant material available in electronic form. Some of the material available is:

- Configuration data examples, which you can copy and paste to use directly in your configuration.
- The tools I wrote for the book, together with documentation and usage examples. Request new features, and I will add them whenever possible.
- The links to all resources mentioned in the book, grouped according to their appearance in chapters. This will help you avoid retyping long links. I intend to maintain the links in working order and to provide copies of resources, should they become unavailable elsewhere.

I hope to expand the companion web site into a useful Apache security resource with a life on its own. Please help by sending your comments and your questions to the email address shown on the web site. I look forward to receiving feedback and shaping the future book releases according to other people's experiences.

Conventions Used in This Book

Throughout this book certain stylistic conventions are followed. Once you are accustomed to them, you will distinguish between comments, commands you need to type, values you need to supply, and so forth.

In some cases, the typeface of the terms in the main text and in code examples will be different. The details of what the different styles (italic, boldface, etc.) mean are described in the following sections.

Programming Conventions

In command prompts shown for Unix systems, prompts that begin with # indicate that you need to be logged in as the *superuser* (*root* username); if the prompt begins with $, then the command can be typed by any user.

Typesetting Conventions

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, usernames, group names, module names, CGI script names, programs, and Unix utilities

Constant width

Indicates commands, options, switches, variables, functions, methods, HTML tags, HTTP headers, status codes, MIME content types, directives in configuration files, the contents of files, code within body text, and the output from commands

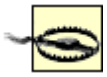
`Constant width bold`

Shows commands or other text that should be typed literally by the user

Constant width italic

Shows text that should be replaced with user-supplied values

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



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Apache Security by Ivan Ristic. Copyright 2005 O'Reilly Media, Inc., 0-596-00724-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

We'd Like to Hear from You

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

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

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

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

Safari Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at http://safari.oreilly.com.

Acknowledgments

This book would not exist, be complete, or be nearly as good if it were not for the work and help of many people. My biggest thanks go to the people believing in the open source philosophy, the Apache developers, and the network and application security communities. It is a privilege to be able to work with you. A book like this cannot exist in isolation. Others have made it possible to write this book by allowing me to stand on their shoulders. Much of their work is referenced throughout the book, but it is impossible to mention it all.

Some people have had a more direct impact on my work. I thank Nathan Torkington and Tatiana Diaz for signing me up with O'Reilly and giving me the opportunity to have my book published by a publisher I respect. My special thanks and gratitude go to my editor, Mary Dageforde, who showed great patience working with me on my drafts. I doubt the book would be nearly as useful, interesting, or accurate without her. My reviewers, Rich Bowen, Dr. Anton Chuvakin, and Sebastian Wolfgarten were there for me to give words of encouragement, very helpful reviews, and a helping hand when it was needed.

I would like to thank Robert Auger, Ryan C. Barnett, Mark Curphey, Jeremiah Grossman, Anders Henke, and Peter Sommerlad for being great people to talk to and work with. My special thanks goes to the merry members of #port80, who were my first contact with the web security community and with whom I've had great fun talking to.

My eternal gratitude goes to my wife Jelena, for inspiring me to lead a better life, and encouraging me to do more and go further. She deserves great credit for putting up with me in the months I did nothing else but work on the book. Finally, I'd like to thank my parents and my family, for bringing me up the way they have, to always seek more but to be at peace with myself over where I am.

Chapter 1. Apache Security Principles

This book contains 12 chapters. Of those, 11 cover the technical issues of securing Apache and web applications. Looking at the number of pages alone it may seem the technical issues represent the most important part of security. But wars are seldom won on tactics alone, and technical issues are just tactics. To win, you need a good overall strategy, and that is the purpose of this chapter. It has the following goals:

- Define security
- Introduce essential security principles
- Establish a common security vocabulary
- Present web application architecture blueprints

The Web Application Architecture Blueprints section offers several different views (user, network, and Apache) of the same problem, with a goal of increasing understanding of the underlying issues.

1.1. Security Definitions

Security can be defined in various ways. One school of thought defines it as reaching the three goals known as the CIA triad:

Confidentiality

Information is not disclosed to unauthorized parties.

Integrity

Information remains unchanged in transit or in storage until it is changed by an authorized party.

Availability

Authorized parties are given timely and uninterrupted access to resources and information.

Another goal, *accountability*, defined as being able to hold users accountable (by maintaining their identity and recording their actions), is sometimes added to the list as a fourth element.

The other main school of thought views security as a continuous process, consisting of phases. Though different people may name and describe the phases in different ways, here is an example of common phases:

Assessment

Analysis of the environment and the system security requirements. During this phase, you create and document a security policy and plans for implementing that policy.

Protection

Implementation of the security plan (e.g., secure configuration, resource protection, maintenance).

Detection

Identification of attacks and policy violations by use of techniques such as monitoring, log analysis, and intrusion detection.

Response

Handling of detected intrusions, in the ways specified by the security plan.

Both lines of thought are correct: one views the static aspects of security and the other views the dynamics. In this chapter, I look at security as a process; the rest of the book covers its static aspects.

Another way of looking at security is as a state of mind. Keeping systems secure is an ongoing battle where one needs be alert and vigilant at all times, and remain one step ahead of adversaries. But you need to come to terms that being 100 percent secure is impossible. Sometimes, we cannot control circumstances, though we do the best we can. Sometimes we slip. Or we may have encountered a smarter adversary. I have found that being humble increases security. If you think you are invincible, chances are you won't be alert to lurking dangers. But if you are aware of

1.2. Web Application Architecture Blueprints

I will now present several different ways of looking at a typical web application architecture. The whole thing is too complex to depict on a single illustration and that's why we need to use the power of abstraction to cope with the complexity. Broken into three different views, the problem becomes easier to manage. The three views presented are the following:

- User view
- Network view
- Apache view

Each view comes with its own set of problems, which need to be addressed one at a time until all problems are resolved. The three views together practically map out the contents of this book. Where appropriate, I will point you to sections where further discussion takes place.

1.2.1. User View

The first view, presented in Figure 1-1, is deceptively simple. Its only purpose is to demonstrate how a typical installation has many types of users. When designing the figure, I chose a typical business installation with the following user classes:

- The public (customers or potential customers)
- Partners
- Staff
- Developers
- Administrators
- Management

Figure 1-1. Web architecture: user view

Chapter 2. Installation and Configuration

Installation is the first step in making Apache functional. Before you begin, you should have a clear idea of the installation's purpose. This idea, together with your paranoia level, will determine the steps you will take to complete the process. The system-hardening matrix (described in Chapter 1) presents one formal way of determining the steps. Though every additional step you make now makes the installation more secure, it also increases the time you will spend maintaining security. Think about it realistically for a moment. If you cannot put in that extra time later, then why bother putting the extra time in now? Don't worry about it too much, however. These things tend to sort themselves out over time: you will probably be eager to make everything perfect in the first couple of Apache installations you do; then, you will likely back off and find a balance among your security needs, the effort required to meet those needs, and available resources.

As a rule of thumb, if you are building a high profile web serverpublic or notalways go for a highly secure installation.

Though the purpose of this chapter is to be a comprehensive guide to Apache installation and configuration, you are encouraged to read others' approaches to Apache hardening as well. Every approach has its unique points, reflecting the personality of its authors. Besides, the opinions presented here are heavily influenced by the work of others. The Apache reference documentation is a resource you will go back to often. In addition to it, ensure you read the Apache Benchmark, which is a well-documented reference installation procedure that allows security to be quantified. It includes a semi-automated scoring tool to be used for assessment.

The following is a list of some of the most useful Apache installation documentation I have encountered:

- Apache Online Documentation (http://httpd.apache.org/docs-2.0/)
- Apache Security Tips (http://httpd.apache.org/docs-2.0/misc/security_tips.html)
- Apache Benchmark (http://www.cisecurity.org/bench_apache.html)
- "Securing Apache: Step-by-Step" by Artur Maj (http://www.securityfocus.com/printable/infocus/1694)
- "Securing Apache 2: Step-by-Step" by Artur Maj (http://www.securityfocus.com/printable/infocus/1786)

2.1. Installation

The installation instructions given in this chapter are designed to apply to both active branches (1.x and 2.x) of the Apache web server running on Linux systems. If you are running some other flavor of Unix, I trust you will understand what the minimal differences between Linux and your system are. The configuration advice given in this chapter works well for non-Unix platforms (e.g., Windows) but the differences in the installation steps are more noticeable:

- Windows does not offer the chroot functionality (see the section Section 2.4) or an equivalent.
- You are unlikely to install Apache on Windows from source code. Instead, download the binaries from the main Apache web site.
- Disk paths are different though the meaning is the same.

2.1.1. Source or Binary

One of the first decisions you will make is whether to compile the server from the source or use a binary package. This is a good example of the dilemma I mentioned at the beginning of this chapter. There is no one correct decision for everyone or one correct decision for you alone. Consider some pros and cons of the different approaches:

- By compiling from source, you are in the position to control everything. You can choose the compile-time options and the modules, and you can make changes to the source code. This process will consume a lot of your time, especially if you measure the time over the lifetime of the installation (it is the only correct way to measure time) and if you intend to use modules with frequent releases (e.g., PHP).
- Installation and upgrade is a breeze when binary distributions are used now that many vendors have tools to have operating systems updated automatically. You exchange some control over the installation in return for not having to do everything yourself. However, this choice means you will have to wait for security patches or for the latest version of your favorite module. In fact, the latest version of Apache or your favorite module may never come since most vendors choose to use one version in a distribution and only issue patches to that version to fix potential problems. This is a standard practice, which vendors use to produce stable distributions.
- The Apache version you intend to use will affect your decision. For example, nothing much happens in the 1.x branch, but frequent releases (with significant improvements) occur in the 2.x branch. Some operating system vendors have moved on to the 2.x branch, yet others remain faithful to the proven and trusted 1.x branch.

The Apache web server is a victim of its own success. The web server from the 1.x branch works so well that many of its users have no need to upgrade. In the long term this situation only slows down progress because developers spend their time maintaining the 1.x branch instead of adding new features to the 2.x branch. Whenever you can, use Apache 2!

This book shows the approach of compiling from the source code since that approach gives us the most power and the flexibility to change things according to our taste. To download the source code, go to http://httpd.apache.org and pick the latest release of the branch you want to use.

2.1.1.1 Downloading the source code

2.2. Configuration and Hardening

Now that you know your installation works, make it more secure. Being brave, we start with an empty configuration file, and work our way up to a fully functional configuration. Starting with an empty configuration file is a good practice since it increases your understanding of how Apache works. Furthermore, the default configuration file is large, containing the directives for everything, including the modules you will never use. It is best to keep the configuration files nice, short, and tidy.

Start the configuration file (*/usr/local/apache/conf/httpd.conf*) with a few general-purpose directives:

```
# location of the web server files
ServerRoot /usr/local/apache
# location of the web server tree
DocumentRoot /var/www/htdocs
# path to the process ID (PID) file, which
# stores the PID of the main Apache process
PidFile /var/www/logs/httpd.pid
# which port to listen at
Listen 80
# do not resolve client IP addresses to names
HostNameLookups Off
```

2.2.1. Setting Up the Server User Account

Upon installation, Apache runs as a user *nobody*. While this is convenient (this account normally exists on all Unix operating systems), it is a good idea to create a separate account for each different task. The idea behind this is that if attackers break into the server through the web server, they will get the privileges of the web server. The intruders will have the same priveleges as in the user account. By having a separate account for the web server, we ensure the attackers do not get anything else free.

The most commonly used username for this account is *httpd*, and some people use *apache*. We will use the former. Your operating system may come pre-configured with an account for this purpose. If you like the name, use it; otherwise, delete it from the system (e.g., using the *userdel* tool) to avoid confusion later. To create a new account, execute the following two commands while running as *root*.

```
# groupadd httpd
# useradd httpd -g httpd -d /dev/null -s /sbin/nologin
```

These commands create a group and a user account, assigning the account the home directory */dev/null* and the shell */sbin/nologin* (effectively disabling login for the account). Add the following two lines to the Apache configuration file *httpd.conf*:

```
User httpd
Group httpd
```

2.2.2. Setting Apache Binary File Permissions

After creating the new user account your first impulse might be to assign ownership over the Apache installation to it. I see that often, but do not do it. For Apache to run on port 80, it must be started by the user *root*. Allowing any other account to have write access to the *httpd* binary would give that account privileges to execute anything as *root*.

This problem would occur, for example, if an attacker broke into the system. Working as the Apache user (*httpd*), he would be able to replace the *httpd* binary with something else and shut the web server down. The administrator, thinking the web server had crashed, would log in and attempt to start it again and would have fallen into the trap of executing a Trojan program.

That is why we make sure only *root* has write access:

```
# chown -R root:root /usr/local/apache
# find /usr/local/apache -type d | xargs chmod 755
# find /usr/local/apache -type f | xargs chmod 644
```

2.3. Changing Web Server Identity

One of the principles of web server hardening is hiding as much information from the public as possible. By extending the same logic, hiding the identity of the web server makes perfect sense. This subject has caused much controversy. Discussions usually start because Apache does not provide facilities to control all of the content provided in the Server header field, and some poor soul tries to influence Apache developers to add it. Because no clear technical reasons support either opinion, discussions continue.

I have mentioned the risks of providing server information in the Server response header field defined in the HTTP standard, so a first step in our effort to avoid this will be to fake its contents. As you will see later, this is often not straightforward, but it can be done. Suppose we try to be funny and replace our standard response "Apache/1.3.30 (Unix)" with "Microsoft-IIS/5.0" (it makes no difference to us that Internet Information Server has a worse security record than Apache; our goal is to hide who we are). An attacker sees this but sees no trace of Active Server Pages (ASP) on the server, and that makes him suspicious. He decides to employ *operating system fingerprinting*. This technique uses the variations in the implementations of the TCP/IP protocol to figure out which operating system is behind an IP address. This functionality comes with the popular network scanner NMAP. Running NMAP against a Linux server will sometimes reveal that the server is not running Windows. Microsoft IIS running on a Linux servernot likely!

There are also differences in the implementations of the HTTP protocol supplied by different web servers. HTTP fingerprinting exploits these differences to determine the make of the web server. The differences exist for the following reasons:

- Standards do not define every aspect of protocols. Some parts of the standard are merely recommendations, and some parts are often intentionally left vague because no one at the time knew how to solve a particular problem so it was left to resolve itself.
- Standards sometimes do not define trivial things.
- Developers often do not follow standards closely, and even when they do, they make mistakes.

The most frequently used example of web server behavior that may allow exploitation is certainly the way Apache treats URL encoded forward slash characters. Try this:

1. Open a browser window, and type in the address **http://www.apachesecurity.net//** (two forward slashes at the end). You will get the home page of the site.
2. Replace the forward slash at the end with %2f (the same character but URL-encoded): **http://www.apachesecurity.net/%2f**. The web server will now respond with a 404 (Not Found) response code!

This happens only if the site runs Apache. In two steps you have determined the make of the web server without looking at the Server header field. Automating this check is easy.

This behavior was so widely and frequently discussed that it led Apache developers to introduce a directive (AllowEncodedSlashes) to the 2.x branch to toggle how Apache behaves. This will not help us much in our continuing quest to fully control the content provided in the Server header field. There is no point in continuing to fight for this. In theory, the only way to hide the identity of the server is to put a reverse proxy (see Chapter 9) in front and instruct it to alter the order of header fields in the response, alter their content, and generally do everything possible to hide the server behind it. Even if someone succeeds at this, this piece of software will be so unique that the attacker will identify the reverse proxy successfully, which is as dangerous as what we have been trying to hide all along.

2.4. Putting Apache in Jail

Even the most secure software installations get broken into. Sometimes, this is because you get the attention of a skilled and persistent attacker. Sometimes, a new vulnerability is discovered, and an attacker uses it before the server is patched. Once an intruder gets in, his next step is to look for local vulnerability and become *superuser*. When this happens, the whole system becomes contaminated, and the only solution is to reinstall everything.

Our aim is to contain the intrusion to just a part of the system, and we do this with the help of the chroot(2) system call. This system call allows restrictions to be put on a process, limiting its access to the filesystem. It works by choosing a folder to become the new filesystem root. Once the system call is executed, a process cannot go back (in most cases, and provided the jail was properly constructed).

The *root* user can almost always break out of jail. The key to building an escape-proof jail environment is not to allow any *root* processes to exist inside the jail. You must also not have a process outside jail running as the same user as a process inside jail. Under some circumstances, an attacker may jump from one process to another and break out of jail. That's one of the reasons why I have insisted on having a separate account for Apache.

The term *chroot* is often interchangeably used with the term *jail*. The term can be used as a verb and noun. If you say Apache is *chrooted*, for example, you are saying that Apache was put in jail, typically via use of the *chroot* binary or the chroot(2) system call. On Linux systems, the meanings of *chroot* and *jail* are close enough. BSD systems have a separate jail( ) call, which implements additional security mechanisms. For more details about the jail( ) call, see the following: http://docs.freebsd.org/44doc/papers/jail/jail.html.

Incorporating the jail mechanism (using either chroot(2) or jail( )) into your web server defense gives the following advantages:

Containment

If the intruder breaks in through the server, he will only be able to access files in the restricted file system. Unable to touch other files, he will be unable to alter them or harm the data in any way.

No shell

Most exploits need shells (mostly */bin/sh*) to be fully operative. While you cannot remove a shell from the operating system, you can remove it from a jail environment.

Limited tool availability

Once inside, the intruder will need tools to progress further. To begin with, he will need a shell. If a shell isn't available he will need to find ways to bring one in from the inside. The intruder will also need a compiler. Many black hat tools are not used as binaries. Instead, these tools are uploaded to the server in source and compiled on the spot. Even many automated attack tools compile programs. The best example is the Apache Slapper Worm (see the sidebar Apache Slapper Worm).

Absence of suid root binaries

Getting out of a jail is possible if you have the privileges of the *root* user. Since all the effort we put into the construction of a jail would be meaningless if we allowed *suid root* binaries, make sure you do not put such files into

Chapter 3. PHP

PHP is the most popular web scripting language and an essential part of the Apache platform. Consequently, it is likely most web application installations will require PHP's presence. However, if your PHP needs are moderate, consider replacing the functionality you need using plain-old CGI scripts. The PHP module is a complex one and one that had many problems in the past.

This chapter will help you use PHP securely. In addition to the information provided here, you may find the following resources useful:

- Security section of the PHP manual (http://www.php.net/manual/en/security.php)
- PHP Security Consortium (http://www.phpsec.org)

3.1. Installation

In this section, I will present the installation and configuration procedures for two different options: using PHP as a module and using it as a CGI. Using PHP as a module is suitable for systems that are dedicated to a single purpose or for sites run by trusted groups of administrators and developers. Using PHP as a CGI (possibly with an execution wrapper) is a better option when users cannot be fully trusted, in spite of its worse performance. (Chapter 6 discusses running PHP over FastCGI which is an alternative approach that can, in some circumstances, provide the speed of the module combined with the privilege separation of a CGI.) To begin with the installation process, download the PHP source code from http://www.php.net.

3.1.1. Using PHP as a Module

When PHP is installed as a module, it becomes a part of Apache and performs all operations as the Apache user (usually *httpd*). The configuration process is similar to that of Apache itself. You need to prepare PHP source code for compilation by calling the *configure* script (in the directory where you unpacked the distribution), at a minimum letting it know where Apache's *apxs* tool resides. The *apxs* tool is used as the interface between Apache and third-party modules:

```
$ ./configure --with-apxs=/usr/local/apache/bin/apxs
$ make
# make install
```

Replace --with-apxs with --with-apxs2 if you are running Apache 2. If you plan to use PHP only from within the web server, it may be useful to put the installation together with Apache. Use the --prefix configuration parameter for that:

```
$ ./configure \
> --with-apxs=/usr/local/apache/bin/apxs \
> --prefix=/usr/local/apache/php
```

In addition to making PHP work with Apache, a command-line version of PHP will be compiled and copied to */usr/local/apache/php/bin/php*. The command-line version is useful if you want to use PHP for general scripting, unrelated to web servers.

The following configuration data makes Apache load PHP when it starts and allows Apache to identify which pages contain PHP code:

```
# Load the PHP module (the module is in
# subdirectory modules/ in Apache 2)
LoadModule php5_module libexec/libphp5.so
# Activate the module (not needed with Apache 2)
AddModule mod_php5.c

# Associate file extensions with PHP
AddHandler application/x-httpd-php .php
AddHandler application/x-httpd-php .php3
AddHandler application/x-httpd-php .inc
AddHandler application/x-httpd-php .class
AddHandler application/x-httpd-php .module
```

I choose to associate several extensions with the PHP module. One of the extensions (*.php3*) is there for backward compatibility. Java class files end in *.class* but there is little chance of clash because these files should never be accessed directly by Apache. The others are there to increase security. Many developers use extensions other than *.php* for their PHP code. These files are not meant to be accessed directly but through an include() statement. Unfortunately, these files are often stored under the web server tree for convenience and anyone who knows their names can request them from the web server. This often leads to a security problem. (This issue is discussed in more detail in Chapter 10 and Chapter 11.)

Next, update the DirectoryIndex directive:

```
DirectoryIndex index.html index.php
```

Finally, place a version of *php.ini* in */usr/local/apache/php/lib/*. A frequent installation error occurs when the configuration file is placed at a wrong location, where it fails to have any effect on the PHP engine. To make sure a

3.2. Configuration

Configuring PHP can be a time-consuming task since it offers a large number of configuration options. The distribution comes with a recommended configuration file *php.ini-recommended*, but I suggest that you just use this file as a starting point and create your own recommended configuration.

3.2.1. Disabling Undesirable Options

Working with PHP you will discover it is a powerful tool, often too powerful. It also has a history of loose default configuration options. Though the PHP core developers have paid more attention to security in recent years, PHP is still not as secure as it could be.

3.2.1.1 register_globals and allow_url_fopen

One PHP configuration option strikes fear into the hearts of system administrators everywhere, and it is called register_globals. This option is off by default as of PHP 4.2.0, but I am mentioning it here because:

- It is dangerous.
- You will sometimes be in a position to audit an existing Apache installation, so you will want to look for this option.
- Sooner or later, you will get a request from a user to turn it on. Do not do this.

I am sure it seemed like a great idea when people were not as aware of web security issues. This option, when enabled, automatically transforms request parameters directly into PHP global parameters. Suppose you had a URL with a name parameter:

```
http://www.apachesecurity.net/sayhello.php?name=Ivan
```

The PHP code to process the request could be this simple:

```
<? echo "Hello $name!"; ?>
```

With web programming being as easy as this, it is no wonder the popularity of PHP exploded. Unfortunately, this kind of functionality led to all sorts of unwanted side effects, which people discovered after writing tons of insecure code. Look at the following code fragment, placed on the top of an administration page:

```
<?
if (isset($admin) =  = false) {
    die "This page is for the administrator only!";
}
?>
```

In theory, the software would set the $admin variable to TRue when it authenticates the user and figures out the user has administration privileges. In practice, appending ?admin=1 to the URL would cause PHP to create the $admin variable where one is absent. And it gets worse.

Another PHP option, allow_url_fopen, allows programmers to treat URLs as files. (This option is still on by default.) People often use data from a request to determine the name of a file to read, as in the following example of an application that expects a parameter to specify the name of the file to execute:

```
http://www.example.com/view.php?what=index.php
```

The application then uses the value of the parameter what directly in a call to the include() language construct:

```
<? include($what) ?>
```

As a result, an attacker can, by sending a path to any file on the system as parameter (for example /etc/passwd), read

3.3. Advanced PHP Hardening

When every little bit of additional security counts, you can resort to modifying PHP. In this section, I present two approaches: one that uses PHP extension capabilities to change its behavior without changing the source code, and another that goes all the way and modifies the PHP source code to add an additional security layer.

3.3.1. PHP 5 SAPI Input Hooks

In PHP, S API stands for *Server Abstraction Application Programming Interface* and is a part of PHP that connects the engine with the environment it is running in. One SAPI is used when PHP is running as an Apache module, a second when running as a CGI script, and a third when running from the command line. Of interest to us are the three input callback hooks that allow changes to be made to the way PHP handles script input data:

input_filter

Called before each script parameter is added to the list of parameters. The hook is given an opportunity to modify the value of the parameter and to accept or refuse its addition to the list.

treat_data

Called to parse and transform script parameters from their raw format into individual parameters with names and values.

default_post_reader

Called to handle a POST request that does not have a handler associated with it.

The input_filter hook is the most useful of all three. A new implementation of this hook can be added through a custom PHP extension and registered with the engine using the sapi_register_input_filter( ) function. The PHP 5 distribution comes with an input filter example (the file *README.input_filter* also available at http://cvs.php.net/co.php/php-src/README.input_filter), which is designed to strip all HTML markup (using the strip_tags( ) function) from script parameters. You can use this file as a starting point for your own extension.

A similar solution can be implemented without resorting to writing native PHP extensions. Using the auto_prepend_file configuration option to prepend input sanitization code for every script that is executed will have similar results in most cases. However, only the direct, native-code approach works in the following situations:

- If you want to enforce a strong site-wide policy that cannot be avoided
- If the operations you want to perform are too slow to be implemented in PHP itself
- When the operations simply require direct access to the PHP engine

3.3.2. Hardened-PHP

Hardened-PHP (http://www.hardened-php.net) is a project that has a goal of remedying some of the shortcomings present in the mainstream PHP distribution. It's a young and promising project led by Stefan Esser. At the time of this writing the author was offering support for the latest releases in both PHP branches (4.x and 5.x). Here are some of the features this patch offers:

Chapter 4. SSL and TLS

Like many other Internet protocols created before it, HTTP was designed under the assumption that data transmission would be secure. This is a perfectly valid assumption; it makes sense to put a separate communication layer in place to worry about issues such as confidentiality and data integrity. Unfortunately, a solution to secure data transmission was not offered at the same time as HTTP. It arrived years later, initially as a proprietary protocol.

By today's standards, the Internet was not a very secure place in the early days. It took us many years to put mechanisms in place for secure communication. Even today, millions of users are using insecure, plaintext communication protocols to transmit valuable, private, and confidential information.

Not taking steps to secure HTTP communication can lead to the following weaknesses:

- Data transmission can be intercepted and recorded with relative ease.
- For applications that require users to authenticate themselves, usernames and passwords are trivial to collect as they flow over the wire.
- User sessions can be hijacked, and attackers can assume users' identities.

Since these are serious problems, the only cases where additional security measures are not required are with a web site where all areas are open to the public or with a web site that does not contain any information worth protecting. Some cases require protection:

- When a web site needs to collect sensitive information from its users (e.g., credit card numbers), it must ensure the communication cannot be intercepted and the information hijacked.
- The communication between internal web applications and intranets is easy to intercept since many users share common network infrastructure (for example, the local area network). Encryption (described later in the chapter) is the only way to ensure confidentiality.
- Mission-critical web applications require a maximum level of security, making encryption a mandatory requirement.

To secure HTTP, the *Secure Sockets Layer* (SSL) protocol is used. This chapter begins by covering cryptography from a practical point of view. You only need to understand the basic principles. We do not need to go into mathematical details and discuss differences between algorithms for most real-life requirements. After documenting various types of encryption, this chapter will introduce SSL and describe how to use the OpenSSL libraries and the *mod_ssl* Apache module. Adding SSL capabilities to the web server is easy, but getting the certificate infrastructure right requires more work. The end of the chapter discusses the impact of SSL on performance and explains how to determine if SSL will represent a bottleneck.

4.1. Cryptography

Cryptography is a mathematical science used to secure storage and transmission of data. The process involves two steps: *encryption* transforms information into unreadable data, and *decryption* converts unreadable data back into a readable form. When cryptography was first used, confidentiality was achieved by keeping the transformation algorithms secret, but people figured out those algorithms. Today, algorithms are kept public and well documented, but they require a secret piece of information; a *key*, to hide and reveal data. Here are three terms you need to know:

Cleartext

Data in the original form; also referred to as *plaintext*

Cipher

The algorithm used to protect data

Ciphertext

Data in the encoded (unreadable) form

Cryptography aims to achieve four goals:

Confidentiality

Protect data from falling into the wrong hands

Authentication

Confirm identities of parties involved in communication

Integrity

Allow recipient to verify information was not modified while in transit

Nonrepudiation

Prevent sender from claiming information was never sent

The point of cryptography is to make it easy to hide (encrypt) information yet make it difficult and time consuming for anyone without the decryption key to decrypt encrypted information.

No one technique or algorithm can be used to achieve all the goals listed above. Instead, several concepts and techniques have to be combined to achieve the full effect. There are four important concepts to cover:

- Symmetric encryption
- Asymmetric encryption

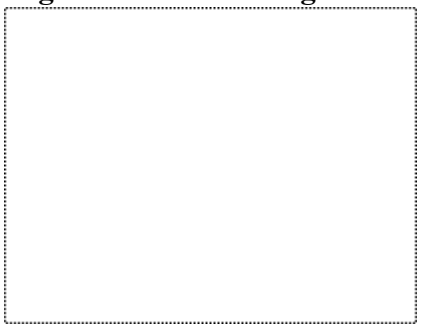
4.2. SSL

Around 1995, Netscape Navigator was dominating the browser market with around a 70 percent share. When Netscape created SSL in 1994, it became an instant standard. Microsoft tried to compete, releasing a technology equivalent, *Private Communication Technology* (PCT), but it had no chance due to Internet Explorer's small market share. It was not until 1996, when Microsoft released Internet Explorer 3, that Netscape's position was challenged.

The first commercial SSL implementation to be released was SSLv2, which appeared in 1994. Version 3 followed in 1995. Netscape also released the SSLv3 reference implementation and worked with the *Internet Engineering Task Force* (IETF) to turn SSL into a standard. The official name of the standard is *Transport Layer Security* (TLS), and it is defined in RFC 2246 (http://www.ietf.org/rfc/rfc2246.txt). TLS is currently at version 1.0, but that version is practically the same as SSLv3.1. In spite of the official standard having a different name everyone continues to call the technology SSL, so that is what I will do, too.

SSL lives above TCP and below HTTP in the *Open Systems Interconnection* (OSI) model, as illustrated in Figure 4-6. Though initially implemented to secure HTTP, SSL now secures many connection-oriented protocols. Examples are SMTP, POP, IMAP, and FTP.

Figure 4-6. SSL belongs to level 6 of the OSI model



In the early days, web hosting required exclusive use of one IP address per hosted web site. But soon hosting providers started running out of IP addresses as the number of web sites grew exponentially. To allow many web sites to share the same IP address, a concept called *name-based virtual hosting* was devised. When it is deployed, the name of the target web site is transported in the Host request header. However, SSL still requires one exclusive IP address per web site. Looking at the OSI model, it is easy to see why. The HTTP request is wrapped inside the encrypted channel, which can be decrypted with the correct server key. But without looking into the request, the web server cannot access the Host header and, therefore, cannot use that information to choose the key. The only information available to the server is the incoming IP address.

Because only a small number of web sites require SSL, this has not been a major problem. Still, a way of upgrading from non-SSL to SSL communication has been designed (see RFC2817 at http://www.ietf.org/rfc/rfc2817.txt).

4.2.1. SSL Communication Summary

SSL is a hybrid protocol. It uses many of the cryptographic techniques described earlier to make communication secure. Every SSL connection consists of essentially two phases:

Handshake phase

During this phase, the server sends the client its certificate (containing its public key) and the client verifies the server's identity using public-key cryptography. In some (relatively infrequent) cases, the server also requires the client to have a certificate, and client verification is also performed. After server (and potentially client) verification is complete, the client and server agree on a common set of encryption protocols and generate a set of private cryptography secret

4.3. OpenSSL

OpenSSL is the open source implementation (toolkit) of many cryptographic protocols. Almost all open source and many commercial packages rely on it for their cryptographic needs. OpenSSL is licensed under a BSD-like license, which allows commercial exploitation of the source code. You probably have OpenSSL installed on your computer if you are running a Unix system. If you are not running a Unix system or you are but you do not have OpenSSL installed, download the latest version from the web site (http://www.openssl.org). The installation is easy:

```
 $ ./config
$ make
# make install
```

Do not download and install a new copy of OpenSSL if one is already installed on your system. You will find that other applications rely on the pre-installed version of OpenSSL. Adding another version on top will only lead to confusion and possible incompatibilities.

OpenSSL is a set of libraries, but it also includes a tool, *openssl*, which makes most of the functionality available from the command line. To avoid clutter, only one binary is used as a façade for many commands supported by OpenSSL. The first parameter to the binary is the name of the command to be executed.

The standard port for HTTP communication over SSL is port 443. To connect to a remote web server using SSL, type something like the following, where this example shows connecting to Thawte's web site:

```
 $ openssl s_client -host www.thawte.com -port 443
```

As soon as the connection with the server is established, the command window is filled with a lot of information about the connection. Some of the information displayed on the screen is quite useful. Near the top is information about the certificate chain, as shown below. A *certificate chain* is a collection of certificates that make a path from the first point of contact (the web site www.thawte.com, in the example above) to a trusted root certificate. In this case, the chain references two certificates, as shown in the following output. For each certificate, the first line shows the information about the certificate itself, and the second line shows information about the certificate it was signed with. Certificate information is displayed in condensed format: the forward slash is a separator, and the uppercase letters stand for certificate fields (e.g., C for country, ST for state). You will get familiar with these fields later when you start creating your own certificates. Here is the certificate chain:

```
 Certificate chain
 0 s:/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting (Pty)
Ltd/OU=Security/CN=www.thawte.com
   i:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
 1 s:/C=ZA/O=Thawte Consulting (Pty) Ltd./CN=Thawte SGC CA
   i:/C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
```

You may be wondering what VeriSign is doing signing a Thawte certificate; Thawte is a CA, after all. VeriSign recently bought Thawte; though they remain as two different business entities, they are sharing a common root certificate.

The details of the negotiated connection with the remote server are near the end of the output:

```
 New, TLSv1/SSLv3, Cipher is EDH-RSA-DES-CBC3-SHA
Server public key is 1024 bit
SSL-Session:
   Protocol  : TLSv1
    Cipher    : EDH-RSA-DES-CBC3-SHA
    Session-ID: 6E9DBBBA986C501A88F0B7ADAFEC6529291C739EB4CC2114EE62036D9B
F04C6E
    Session-ID-ctx:
    Master-Key: 0D90A33260738C7B8CBCC1F2A5DC3BE79D9D4E2FC7C649E5A541594F37
61CE7046E7F5034933A6F09D7176E2B0E11605
    Key-Arg   : None
    Krb5 Principal: None
    Start Time: 1090586684
    Timeout   : 300 (sec)
    Verify return code: 20 (unable to get local issuer certificate)
```

4.4. Apache and SSL

If you are using Apache from the 2.x branch, the support for SSL is included with the distribution. For Apache 1, it is a separate download of one of two implementations. You can use *mod_ssl* (http://www.modssl.org) or Apache-SSL (http://www.apache-ssl.org). Neither of these two web sites discusses why you would choose one instead of the other. Historically, *mod_ssl* was created out of Apache-SSL, but that was a long time ago and the two implementations have little in common (in terms of source code) now. The *mod_ssl* implementation made it into Apache 2 and is more widely used, so it makes sense to make it our choice here.

Neither of these implementations is a simple Apache module. The Apache 1 programming interface does not provide enough functionality to support SSL, so *mod_ssl* and Apache-SSL rely on modifying the Apache source code during installation.

4.4.1. Installing mod_ssl

To add SSL to Apache 1, download and unpack the *mod_ssl* distribution into the same top folder where the existing Apache source code resides. In my case, this is */usr/local/src*. I will assume you are using Apache Version 1.3.31 and *mod_ssl* Version 2.8.19-1.3.31:

```
$ cd /usr/local/src
$ wget -q http://www.modssl.org/source/mod_ssl-2.8.19-1.3.31.tar.gz
$ tar zxvf mod_ssl-2.8.19-1.3.31.tar.gz
$ cd mod_ssl-2.8.19-1.3.31
$ ./configure --with-apache=../apache_1.3.31
```

Return to the Apache source directory (cd ../apache_1.3.31) and configure Apache, adding a --enable-module=ssl switch to the *configure* command. Proceed to compile and install Apache as usual:

```
$ ./configure --prefix=/usr/local/apache --enable-module=ssl
$ make
# make install
```

Adding SSL to Apache 2 is easier as you only need to add a --enable-ssl switch to the configure line. Again, recompile and reinstall. I advise you to look at the configuration generated by the installation (in *httpd.conf* for Apache 1 or *ssl.conf* for Apache 2) and familiarize yourself with the added configuration options. I will cover these options in the following sections.

4.4.2. Generating Keys

Once SSL is enabled, the server will not start unless a private key and a certificate are properly configured. Private keys are commonly protected with passwords (also known as *passphrases*) to add additional protection for the keys. But when generating a private key for a web server, you are likely to leave it unprotected because a password-protected private key would require the password to be manually typed every time the web server is started or reconfigured. This sort of protection is not realistic. It is possible to tell Apache to ask an external program for a passphrase (using the SSLPassPhraseDialog directive), and some people use this option to keep the private keys encrypted and avoid manual interventions. This approach is probably slightly more secure but not much. To be used to unlock the private key, the passphrase must be available in cleartext. Someone who is after the private key is likely to be determined enough to continue to look for the passphrase.

The following generates a nonprotected, 1,024-bit server private key using the RSA algorithm (as instructed by the genrsa command) and stores it in *server.key*:

```
# cd /usr/local/apache/conf
# mkdir ssl
# cd ssl
# openssl genrsa -out server.key 1024
Generating RSA private key, 1024 bit long modulus
.....................................++++++
.........................++++++
e is 65537 (0x10001)
```

4.5. Setting Up a Certificate Authority

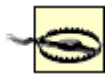
If you want to become a CA, everything you need is included in the OpenSSL toolkit. This step is only feasible in a few high-end cases in which security is critical and you need to be in full control of the process. T he utilities provided with OpenSSL will perform the required cryptographic computations and automatically track issued certificates using a simple, file-based database. To be honest, the process can be cryptic (no pun intended) and frustrating at times, but that is because experts tend to make applications for use by other experts. Besides, polishing applications is not nearly as challenging as inventing something new. Efforts are under way to provide more user-friendly and complete solutions. Two popular projects are:

OpenCA (http://www.openca.org/openca/)

Aims to be a robust out-of-the-box CA solution

TinyCA (http://tinyca.sm-zone.net)

Aims to serve only as an OpenSSL frontend



The most important part of CA operation is making sure the CA's private key remains private. If you are serious about your certificates, keep the CA files on a computer that is not connected to any network. You can use any old computer for this purpose. Remember to backup the files regularly.

After choosing a machine to run the CA operations on, remove the existing OpenSSL installation. Unlike what I suggested for web servers, for CA operation it is better to download the latest version of the OpenSSL toolkit from the main distribution site. The installation process is simple. You do not want the toolkit to integrate into the operating system (you may need to move it around later), so specify a new location for it. The following will configure, compile, and install the toolkit to */opt/openssl*:

```
$ ./configure --prefix=/opt/openssl
$ make
$ make test
# make install
```

Included with the OpenSSL distribution is a convenience tool *CA.pl* (called *CA.sh* or *CA* in some distributions), which simplifies CA operations. The *CA.pl* tool was designed to perform a set of common operations with little variation as an alternative to knowing the OpenSSL commands by heart. This is particularly evident with the usage of default filenames, designed to be able to transition seamlessly from one step (e.g., generate a CSR) to another (e.g., sign the CSR).

Before the CA keys are generated, there are three things you may want to change:

- By default, the generated CA certificates are valid for one year. This is way too short, so you should increase this to a longer period (for example, 10 years) if you intend to use the CA (root) certificate in production. At the beginning of the *CA.pl* file, look for the line $DAYS="-days 365", and change the number of days from 365 to a larger number, such as 3650 for 10 years. This change will affect only the CA certificate and not the others you will generate later.
- The CA's key should be at least 2,048 bits long. Sure, 1024-bit keys are considered strong today, but no one knows what will happen in 10 years' time. To use 2,048-bit keys you will have to find (in *CA.pl*) the part of the code where the CA's certificate is generated (search for "Making CA certificate") and replace $REQ

4.6. Performance Considerations

SSL has a reputation for being slow. This reputation originated in its early days when it was slow compared to the processing power of computers. Things have improved. Unless you are in charge of a very large web installation, I doubt you will experience performance problems with SSL.

4.6.1. OpenSSL Benchmark Script

Since OpenSSL comes with a benchmark script, we do not have to guess how fast the cryptographic functions SSL requires are. The script will run a series of computing-intensive tests and display the results. Execute the script via the following:

```
$ openssl speed
```

The following results were obtained from running the script on a machine with two 2.8 GHz Pentium 4 Xeon processors. The benchmark uses only one processor for its measurements. In real-life situations, both processors will be used; therefore, the processing capacity on a dual server will be twice as large.

The following are the benchmark results of one-way and symmetrical algorithms:

```
 type           16 bytes     64 bytes    256 bytes  1024 bytes  8192 bytes
md2             1841.78k     3965.80k     5464.83k    5947.39k    6223.19k
md4            17326.58k    55490.11k   138188.97k  211403.09k  263528.45k
md5            12795.17k    41788.59k   117776.81k  234883.07k  332759.04k
hmac(md5)       8847.31k    32256.23k   101450.50k  217330.69k  320913.41k
sha1            9529.72k    29872.66k    75258.54k  117943.64k  141710.68k
rmd160         10551.10k    31148.82k    62616.23k  116250.38k  101944.89k
rc4            90858.18k   102016.45k   104585.22k  105199.27k  105250.82k
des cbc        45279.25k    47156.76k    47537.41k   47827.29k   47950.51k
des ede3       17932.17k    18639.27k    18866.43k   18930.35k   18945.37k
rc2 cbc        11813.34k    12087.81k    12000.34k   12156.25k   12113.24k
blowfish cbc   80290.79k    83618.41k    84170.92k   84815.87k   84093.61k
cast cbc       30767.63k    32477.40k    32840.53k   32925.35k   32863.57k
aes-128 cbc    51152.56k    52996.52k    54039.55k   54286.68k   53947.05k
aes-192 cbc    45540.74k    46613.01k    47561.56k   47818.41k   47396.18k
aes-256 cbc    40427.22k    41204.46k    42097.83k   42277.21k   42125.99k
```

Looking at the first column of results for RC4 (a widely used algorithm today), you can see that it offers a processing speed of 90 MBps, and that is using one processor. This is so fast that it is unlikely to create a processing bottleneck.

The benchmark results obtained for asymmetrical algorithms were:

```
                  sign    verify    sign/s verify/s
rsa  512 bits   0.0008s   0.0001s   1187.4  13406.5
rsa 1024 bits   0.0041s   0.0002s    242.0   4584.5
rsa 2048 bits   0.0250s   0.0007s     40.0   1362.2
rsa 4096 bits   0.1705s   0.0026s      5.9    379.0

                  sign    verify    sign/s verify/s
dsa  512 bits   0.0007s   0.0009s   1372.6   1134.0
dsa 1024 bits   0.0021s   0.0026s    473.9    389.9
dsa 2048 bits   0.0071s   0.0087s    141.4    114.4
```

These benchmarks are slightly different. Since asymmetric encryption is not used for data transport but instead is used only during the initial handshake for authentication validation, the results show how many signing operations can be completed in a second. Assuming 1,024-bit RSA keys are used, the processor we benchmarked is capable of completing 242 signing operations per second. Indeed, this seems much slower than our symmetrical encryption tests.

Asymmetrical encryption methods are used at the beginning of each SSL session. The results above show that the processor tested above, when 1,024-bit RSA keys are used, is limited to accepting 242 new connections every second. A large number of sites have nowhere near this number of new connections in a second but this number is not out of the reach of busier e-commerce operations.

Chapter 5. Denial of Service Attacks

A *denial of service* (DoS) *attack* is an attempt to prevent legitimate users from using a service. This is usually done by consuming all of a resource used to provide the service. The resource targeted is typically one of the following:

- CPU
- Operating memory (RAM)
- Bandwidth
- Disk space

Sometimes, a less obvious resource is targeted. Many applications have fixed length internal structures and if an attacker can find a way to populate all of them quickly, the application can become unresponsive. A good example is the maximum number of Apache processes that can exist at any one time. Once the maximum is reached, new clients will be queued and not served.

DoS attacks are not unique to the digital world. They existed many years before anything digital was created. For example, someone sticking a piece of chewing gum into the coin slot of a vending machine prevents thirsty people from using the machine to fetch a refreshing drink.

In the digital world, DoS attacks can be acts of vandalism, too. They are performed for fun, pleasure, or even financial gain. In general, DoS attacks are a tough problem to solve because the Internet was designed on a principle that everyone plays by the rules.

You can become a victim of a DoS attack for various reasons:

Bad luck

In the worst case, you may be at the wrong place at the wrong time. Someone may think your web site is a good choice for an attack, or it may simply be the first web site that comes to mind. He may decide he does not like you personally and choose to make your life more troubled. (This is what happened to Steve Gibson, of http://www.grc.com fame, when a 13-year-old felt offended by the "script kiddies" term he used.)

Controversial content

Some may choose to attack you because they do not agree with the content you are providing. Many people believe disrupting your operation is acceptable in a fight for their cause. Controversial subjects such as the right to choose, globalization, and politics are likely to attract their attention and likely to cause them to act.

Unfair competition

In a fiercely competitive market, you may end up against competitors who will do anything to win. They may constantly do small things that slow you down or go as far as to pay someone to attack your resources.

Controversy over a site you host

5.1. Network Attacks

Network attacks are the most popular type of attack because they are easy to execute (automated tools are available) and difficult to defend against. Since these attacks are not specific to Apache, they fall outside the scope of this book and thus they are not covered in detail in the following sections. As a rule of thumb, only your upstream provider can defend you from attacks performed on the network level. At the very least you will want your provider to cut off the attacks at their routers so you do not have to pay for the bandwidth incurred by the attacks.

5.1.1. Malformed Traffic

The simplest network attacks target weaknesses in implementations of the TCP/IP protocol. Some implementations are not good at handling error conditions and cause systems to crash or freeze. Some examples of this type of attack are:

- Sending very large *Internet Control Message Protocol* (ICMP) packets. This type of attack, known as the *Ping of death*, caused crashes on some older Windows systems.
- Setting invalid flags on TCP/IP packets.
- Setting the destination and the source IP addresses of a TCP packet to the address of the attack target (*Land attack*).

These types of attacks have only historical significance, since most TCP/IP implementations are no longer vulnerable.

5.1.2. Brute-Force Attacks

In the simplest form, an effective network attack can be performed from a single host with a fast Internet connection against a host with a slower Internet connection. By using brute force, sending large numbers of traffic packets creates a *flood attack* and disrupts target host operations. The concept is illustrated in Figure 5-1.

Figure 5-1. Brute-force DoS attack

At the same time, this type of attack is the easiest to defend against. All you need to do is to examine the incoming traffic (e.g., using a packet sniffer like *tcpdump*), discover the IP address from which the traffic is coming from, and instruct your upstream provider to block the address at their router.

At first glance, you may want to block the attacker's IP address on your own firewall but that will not help. The purpose of this type of attack is to saturate the Internet connection. By the time a packet reaches your router (or server), it has done its job.

Be prepared and have contact details of your upstream provider (or server hosting company) handy. Larger companies have many levels of support and quickly reaching someone knowledgable may be difficult. Research telephone numbers in advance. If you can, get to know your administrators before you need their help.

5.2. Self-Inflicted Attacks

Administrators often have only themselves to blame for service failure. Leaving a service configured with default installation parameters is asking for trouble. Such systems are very susceptible to DoS attacks and a simple traffic spike can imbalance them.

5.2.1. Badly Configured Apache

One thing to watch for with Apache is memory usage. Assuming Apache is running in prefork mode, each request is handled by a separate process. To serve one hundred requests at one time, a hundred processes are needed. The maximum number of processes Apache can create is controlled with the MaxClients directive, which is set to 256 by default. This default value is often used in production and that can cause problems if the server cannot cope with that many processes.

Figuring out the maximum number of Apache processes a server can accommodate is surprisingly difficult. On a Unix system, you cannot obtain precise figures on memory utilization. The best thing we can do is to use the information we have, make assumptions, and then simulate traffic to correct memory utilization issues.

Looking at the output of the *ps* command, we can see how much memory a single process takes (look at the RSZ column as it shows the amount of physical memory in use by a process):

```
# ps -A -o pid,vsz,rsz,command
  PID   VSZ  RSZ COMMAND
 3587  9580 3184 /usr/local/apache/bin/httpd
 3588  9580 3188 /usr/local/apache/bin/httpd
 3589  9580 3188 /usr/local/apache/bin/httpd
 3590  9580 3188 /usr/local/apache/bin/httpd
 3591  9580 3188 /usr/local/apache/bin/httpd
 3592  9580 3188 /usr/local/apache/bin/httpd
```

In this example, each Apache instance takes 3.2 MB. Assuming the default Apache configuration is in place, this server requires 1 GB of RAM to reach the peak capacity of serving 256 requests in parallel, and this is only assuming additional memory for CGI scripts and dynamic pages will not be required.

Most web servers do not operate at the edge of their capacity. Your initial goal is to limit the number of processes to prevent server crashes. If you set the maximum number of processes to a value that does not make full use of the available memory, you can always change it later when the need for more processes appears.

Do not be surprised if you see systems with very large Apache processes. Apache installations with a large number of virtual servers and complex configurations require large amounts of memory just to store the configuration data. Apache process sizes in excess of 30 MB are common.

So, suppose you are running a busy, shared hosting server with hundreds of virtual hosts, the size of each Apache process is 30 MB, and some of the sites have over 200 requests at the same time. How much memory do you need? Not as much as you may think.

Most modern operating systems (Linux included) have a feature called *copy-on-write*, and it is especially useful in cases like this one. When a process forks to create a new process (such as an Apache child), the kernel allocates the required amount of memory to accommodate the size of the process. However, this will be virtual memory (of which there is plenty), not physical memory (of which there is little). Memory locations of both processes will point to the same physical memory location. Only when one of the processes attempts to make changes to data will the kernel separate the two memory locations and give each process its own physical memory segment. Hence, the name copy-on-write.

5.3. Traffic Spikes

A sudden spike in the web server traffic can have the same effect as a DoS attack. A well-configured server will cope with the demand, possibly slowing down a little or refusing some clients. If the server is not configured properly, it may crash.

Traffic spikes occur for many reasons, and some of them may be normal. A significant event will cause people to log on and search for more information on the subject. If a site often takes a beating in spite of being properly configured, perhaps it is time to upgrade the server or the Internet connection.

The following sections describe the causes and potential solutions for traffic spikes.

5.3.1. Content Compression

If you have processing power to spare but not enough bandwidth, you might exchange one for the other, making it possible to better handle traffic spikes. Most modern browsers support content compression automatically: pages are compressed before they leave the server and decompressed after they arrive at the client. The server will know the client supports compression when it receives a request header such as this one:

```
Accept-Encoding: gzip,deflate
```

Content compression makes sense when you want to save the bandwidth, and when the clients have slow Internet connections. A 40-KB page may take eight seconds to download over a modem. If it takes the server a fraction of a second to compress the page to 15 KB (good compression ratios are common with HTML pages), the 25-KB length difference will result in a five-second acceleration. On the other hand, if your clients have fast connection speeds (e.g., on local networks), there will be no significant download time reduction.

For Apache 1, *mod_gzip* (http://www.schroepl.net/projekte/mod_gzip/) is used for content compression. For Apache 2, *mod_deflate* does the same and is distributed with the server. However, compression does not have to be implemented on the web server level. It can work just as well in the application server (e.g., PHP; see http://www.php.net/zlib) or in the application.

5.3.2. Bandwidth Attacks

Bandwidth stealing (also known as *hotlinking*) is a common problem on the Internet. It refers to the practice of rogue sites linking directly to files (often images) residing on other sites (victims). To users, it looks like the files are being provided by the rogue site, while the owner of the victim site is paying for the bandwidth.

One way to deal with this is to use *mod_rewrite* to reject all requests for images that do not originate from our site. We can do this because browsers send the address of the originating page in the Referer header field of every request. Valid requests contain the address of our site in this field, and this allows us to reject everything else.

```
 # allow empty referrers, for when a user types the URL directly
RewriteCond %{HTTP_REFERER} !^$

# allow users coming from apachesecurity.net
RewriteCond %{HTTP_REFERER} !^http://www\.apachesecurity\.net [nocase]

# only prevent images from being hotlinked - otherwise
# no one would be able to link to the site at all!
RewriteRule (\.gif|\.jpg|.\png|\.swf)$ $0 [forbidden]
```

Some people have also reported attacks by competitors with busier sites, performed by embedding many invisible tiny (typically 1x1 pixel) frames pointing to their sites. Innocent site visitors would visit the competitor's web site and open an innocent-looking web page. That "innocent" web page would then open dozens of connections to the target web site, usually targeting large images for download. And all this without the users realizing what is happening. Luckily, these attacks can be detected and prevented with the *mod_rewrite* trick described above.

5.3.3. Cyber-Activism

5.4. Attacks on Apache

With other types of attacks being easy, almost trivial, to perform, hardly anyone bothers attacking Apache directly. Under some circumstances, Apache-level attacks can be easier to perform because they do not require as much bandwidth as other types of attacks. Some Apache-level attacks can be performed with as few as a dozen bytes.

Less-skilled attackers will often choose this type of attack because it is so obvious.

5.4.1. Apache Vulnerabilities

Programming errors come in different shapes. Many have security implications. A programming error that can be exploited to abuse system resources should be classified as a vulnerability. For example, in 1998, a programming error was discovered in Apache: specially crafted small-sized requests caused Apache to allocate large amounts of memory. For more information, see:
"YA Apache DoS Attack," discovered by Dag-Erling Smørgrav (
http://marc.theaimsgroup.com/?l=bugtraq&m=90252779826784&w=2)

More serious vulnerabilities, such as nonexploitable buffer overflows, can cause the server to crash when attacked. (Exploitable buffer overflows are not likely to be used as DoS attacks since they can and will be used instead to compromise the host.)

When Apache is running in a prefork mode as it usually is, there are many instances of the server running in parallel. If a child crashes, the parent process will create a new child. The attacker will have to send a large number of requests constantly to disrupt the operation.

A crash will prevent the server from logging the offending request since logging takes place in the last phase of request processing. The clue that something happened will be in the error log, as a message that a segmentation fault occurred. Not all segmentation faults are a sign of attack though. The server can crash under various circumstances (typically due to bugs), and some vendor-packaged servers crash quite often. Several ways to determine what is causing the crashes are described in Chapter 8.

In a multithreaded (not prefork) mode of operation, there is only one server process. A crash while processing a request will cause the whole server to go down and make it unavailable. This will be easy to detect because you have server monitoring in place or you start getting angry calls from your customers.

Vulnerabilities are easy to resolve in most cases: you need to patch the server or upgrade to a version that fixes the problem. Things can be unpleasant if you are running a vendor-supplied version of Apache, and the vendor is slow in releasing the upgrade.

5.4.2. Brute-Force Attacks

Any of the widely available web server load-testing tools can be used to attack a web server. It would be a crude, visible, but effective attack nevertheless. One such tool, *ab* (short for Apache Benchmark), is distributed with Apache. To perform a simple attack against your own server, execute the following, replacing the URL with the URL for your server.

```
$ /usr/local/apache/bin/ab -n 1000 -c 100  http://www.yourserver.com/
```

Choose the concurrency level (the -c switch) to be the same as or larger than the maximum number of Apache processes allowed (MaxClients). The slower the connection to the server, the more effect the attack will have. You will probably find it difficult to perform the attack from the local network.

To defend against this type of attack, first identify the IP address the attacker is coming from and then deny it access

5.5. Local Attacks

Not all attacks come from the outside. Consider the following points:

- In the worst case scenario (from the security point of view), you will have users with shell access and access to a compiler. They can upload files and compile programs as they please.
- Suppose you do not allow shell access but you do allow CGI scripts. Your users can execute scripts, or they can compile binaries and upload and execute them. Similarly, if users have access to a scripting engine such as PHP, they may be able to execute binaries on the system.
- Most users are not malicious, but accidents do happen. A small programming mistake can lead to a server-wide problem. The wider the user base, the greater the chances of having a user that is just beginning to learn programming. These users will typically treat servers as their own workstations.
- Attackers can break in through an account of a legitimate user, or they can find a weakness in the application layer and reach the server through that.

Having a malicious user on the system can have various consequences, but in this chapter, we are concerned only with the DoS attacks. What can such a user do? As it turns out, most systems are not prepared to handle DoS attacks, and it is easy to bring the server down from the inside via the following possibilites:

Process creation attacks

A *fork bomb* is a program that creates copies of itself in an infinite loop. The number of processes grows exponentially and fills the process table (which is limited in size), preventing the system from creating new processes. Processes that were active prior to the fork bomb activation will still be active and working, but an administrator will have a difficult time logging in to kill the offending program. You can find more information about fork bombs at http://www.voltronkru.com/library/fork.html.

Memory allocation attacks

A *malloc bomb* is a program that allocates large amounts of memory. Trying to accommodate the program, the system will start swapping, use up all of its swap space, and finally crash.

Disk overflow attacks

Disk overflow attacks require a bit more effort and thought than the previous two approaches. One attack would create a large file (as easy as cat /dev/zero > /tmp/log). Creating a very large number of small files, and using up the inodes reserved for the partition, will have a similar effect on the system, i.e., prevent it from creating new files.

To keep the system under control, you need to:

- Put user files on a separate partition to prevent them from affecting system partitions.
- Use filesystem quotas. (A good tutorial can be found in the Red Hat 9 manual at http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/ch-disk-quotas.html.)

5.6. Traffic-Shaping Modules

Traffic shaping is a technique that establishes control over web server traffic. Many Apache modules perform traffic shaping, and their goal is usually to slow down a (client) IP address or to control the bandwidth consumption on the per-virtual host level. As a side effect, these modules can be effective against certain types of DoS attacks. The following are some of the more popular traffic-shaping modules:

- *mod_throttle* (http://www.snert.com/Software/mod_throttle/)
- *mod_bwshare* (http://www.topology.org/src/bwshare/)
- *mod_limitipconn* (http://dominia.org/djao/limitipconn.html)

One module is designed specifically as a remedy for Apache DoS attacks:

- *mod_dosevasive* (http://www.nuclearelephant.com/projects/dosevasive/)

The *mod_dosevasive* module will allow you to specify a maximal number of requests executed by the same IP address against one Apache child. If the threshold is reached, the IP address is blacklisted for a time period you specify. You can send an email message or execute a system command (to talk to a firewall, for example) when that happens.

The *mod_dosevasive* module is not as good as it could be because it does not use shared memory to keep information about previous requests persistent. Instead, the information is kept with each child. Other children know nothing about abuse against one of them. When a child serves the maximum number of requests and dies, the information goes with it.

Blacklisting IP addresses can be dangerous. An attempt to prevent DoS attacks can become a self-inflicted DoS attack because users in general do not have unique IP addresses. Many users browse through proxies or are hidden behind a *network address translation* (NAT) system. Blacklisting a proxy will cause all users behind it to be blacklisted. If you really must use one of the traffic-shaping techniques that uses the IP address of the client for that purpose, do the following:

1. Know your users (before you start the blacklist operation).
2. See how many are coming to your web site through a proxy, and never blacklist its IP address.
3. In the blacklisting code, detect HTTP headers that indicate the request came through a proxy (HTTP_FORWARDED, HTTP_X_FORWARDED, HTTP_VIA) and do not blacklist those.
4. Monitor and verify each violation.

5.7. DoS Defense Strategy

With some exceptions (such as with vulnerabilities that can be easily fixed) DoS attacks are very difficult to defend against. The main problem remains being able to distinguish legitimate requests from requests belonging to an attack.

The chapter concludes with a strategy for handling DoS attacks:

1. Treat DoS attacks as one of many possible risks. Your assessment about the risk will influence the way you prepare your defense.
2. Learn about the content hosted on the server. It may be possible to improve software characteristics (and make it less susceptible to DoS attacks) in advance.
3. Determine what you will do when various types of attacks occur. For example, have the contact details of your upstream provider ready.
4. Monitor server operation to detect attacks as soon as possible.
5. Act promptly when attacked.
6. If attacks increase, install automated tools for defense.

Chapter 6. Sharing Servers

The remainder of this book describes methods for preventing people from compromising the Apache installation. In this chapter, I will discuss how to retain control and achieve reasonable security in spite of giving your potential adversaries access to the server. Rarely will you be able to keep the server to yourself. Even in the case of having your own private server, there will always be at least one friend who is in need of a web site. In most cases, you will share servers with fellow administrators, developers, and other users.

You can share server resources in many different ways:

- Among a limited number of selected users (e.g., developers)
- Among a large number of users (e.g., students)
- Massive shared hosting, or sharing among a very large number of users

Though each of these cases has unique requirements, the problems and aims are always the same:

- You cannot always trust other people.
- You must protect system resources from users.
- You must protect users from each other.

As the number of users increases, keeping the server secure becomes more difficult. There are three factors that are a cause for worry: error, malice, and incompetence. Anyone, including you and me, can make a mistake. The only approach that makes sense is to assume we will and to design our systems to fail gracefully.

6.1. Sharing Problems

Many problems can arise when resources are shared among a group of users:

- File permission problems
- Dynamic-content problems
- Resource-sharing problems on the server
- Domain name-sharing problems (which affect cookies and authentication)
- Information leaks on execution boundaries

6.1.1. File Permission Problems

When a server is shared among many users, it is common for each user to have a seperate account. Users typically work with files directly on the system (through a shell of some kind) or manipulate files using the FTP protocol. Having all users use just one web server causes the first and most obvious issue: problems with file permissions.

Users expect and require privacy for their files. Therefore, file permissions are used to protect files from being accessed by other users. Since Apache is effectively just another user (I assume *httpd* in this book), allowances must be made for Apache to access the files that are to be published on the Web. This is a common requirement. Other daemons (Samba and FTPD come to mind) fulfill the same requirements. These daemons initially run as *root* and switch to the required user once the user authenticates. From that moment on, the permission problems do not exist since the process that is accessing the files is the owner of the files.

When it comes to Apache, however, two facts complicate things. For one, running Apache as *root* is heavily frowned upon and normally not possible. To run Apache as *root*, you must compile from the source, specifying a special compile-time option. Without this, the main Apache process cannot change its identity into another user account. The second problem comes from HTTP being a stateless protocol. When someone connects to an FTP server, he stays connected for the length of the session. This makes it easy for the FTP daemon to keep one dedicated process running during that time and avoid file permission problems. But with any web server, one process accessing files belonging to user X now may be accessing the files belonging to user Y the next second.

Like any other user, Apache needs read access for files in order to serve them and execute rights to execute scripts. For folders, the minimum privilege required is execute, though read access is needed if you want directory listings to work. One way to achieve this is to give the required access rights to the world, as shown in the following example:

```
# chmod 701 /home/ivanr
# find /home/ivanr/public_html -type f | xargs chmod 644
# find /home/ivanr/public_html -type d | xargs chmod 755
```

But this is not very secure. Sure, Apache would get the required access, but so would anyone else with a shell on the server. Then there is another problem. Users' public web folders are located inside their home folders. To get into the public web folder, limited access must be allowed to the home folder as well. Provided only the execute privilege is given, no one can list the contents of the home folder, but if they can guess the name of a private file, they will be able to access it in most cases. In a way, this is like having a hole in the middle of your living room and having to think about not falling through every day. A safer approach is to use group membership. In the following example, it is assumed Apache is running as user *httpd* and group *httpd*, as described in Chapter 2:

```
# chgrp httpd /home/ivanr
# chmod 710 /home/ivanr
# chown -R ivanr:httpd /home/ivanr/public_html
```

6.2. Distributing Configuration Data

Apache configuration data is typically located in one or more files in the *conf/* folder of the distribution, where only the *root* user has access. Sometimes, it is necessary or convenient to distribute configuration data, and there are two reasons to do so:

- Distributed configuration files can be edited by users other than the *root* user.
- Configuration directives in distributed configuration files are resolved on every request, which means that any changes take effect immediately without having to have Apache restarted.

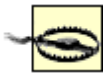
If you trust your developers and want to give them more control over Apache or if you do not trust a junior system administrator enough to give her control over the whole machine, you can choose to give such users full control only over Apache configuration and operation. Use Sudo (http://www.courtesan.com/sudo/) to configure your system to allow non-*root* users to run some commands as *root*.

Apache distributes configuration data by allowing specially-named files, *.htaccess* by default, to be placed together with the content. The name of the file can be changed using the AccessFileName directive, but I do not recommend this. While serving a request for a file somewhere, Apache also looks to see if there are *.htaccess* files anywhere on the path. For example, if the full path to the file is */var/www/htdocs/index.html*, Apache will look for the following (in order):

```
 /.htaccess
/var/.htaccess
/var/www/.htaccess
/var/www/htdocs/.htaccess
```

For each *.htaccess* file found, Apache merges it with the existing configuration data. All *.htaccess* files found are processed, and it continues to process the request. There is a performance penalty associated with Apache looking for access files everywhere. Therefore, it is a good practice to tell Apache you make no use of this feature in most directories (see below) and to enable it only where necessary.

The syntax of access file content is the same as that in *httpd.conf*. However, Apache understands the difference between the two, and understands that some access files will be maintained by people who are not to be fully trusted. This is why administrators are given a choice as to whether to enable access files and, if such files are enabled, which of the Apache features to allow in them.



Another way to distribute Apache configuration is to include other files from the main *httpd.conf* file using the Include directive. This is terribly insecure! You have no control over what is written in the included file, so whoever holds write access to that file holds control over Apache.

Access file usage is controlled with the AllowOverride directive. I discussed this directive in Chapter 2, where I recommended a None setting by default:

```
 <Directory />
    AllowOverride None
</Directory>
```

This setting tells Apache not to look for *.htaccess* files and gives maximum performance and maximum security. To

6.3. Securing Dynamic Requests

Securing dynamic requests is a problem facing most Apache administrators. In this section, I discuss how to enable CGI and PHP scripts and make them run securely and with acceptable performance.

6.3.1. Enabling Script Execution

Because of the inherent danger executable files introduce, execution should always be disabled by default (as discussed in Chapter 2). Enable execution in a controlled manner and only where necessary. Execution can be enabled using one of four main methods:

- Using the ScriptAlias directive
- Explicitly by configuration
- Through server-side includes
- By assigning a handler, type, or filter

6.3.1.1 ScriptAlias versus script enabling by configuration

Using ScriptAlias is a quick and dirty approach to enabling script execution:

```
ScriptAlias /cgi-script/ /home/ivanr/cgi-bin/
```

Though it works fine, this approach can be dangerous. This directive creates a virtual web folder and enables CGI script execution in it but leaves the configuration of the actual folder unchanged. If there is another way to reach the same folder (maybe it's located under the web server tree), visitors will be able to download script source code. Enabling execution explicitly by configuration will avoid this problem and help you understand how Apache works:

```
<Directory /home/ivanr/public_html/cgi-bin>
    Options +ExecCGI
    SetHandler cgi-script
</Directory>
```

6.3.1.2 Server-side includes

Execution of server-side includes (SSIs) is controlled via the Options directive. When the Options +Includes syntax is used, it allows the exec element, which in turn allows operating system command execution from SSI files, as in:

```
<!--#exec cmd="ls" -->
```

To disable command execution but still keep SSI working, use Options +IncludesNOEXEC.

6.3.1.3 Assigning handlers, types, or filters

For CGI script execution to take place, two conditions must be fulfilled. Apache must know execution is what is wanted (for example through setting a handler via SetHandler cgi-script), and script execution must be enabled as a special security measure. This is similar to how an additional permission is required to enable SSIs. Special permissions are usually not needed for other (non-CGI) types of executable content. Whether they are is left for the modules' authors to decide, so it may vary. For example, to enable PHP, it is enough to have the PHP module installed and to assign a handler to PHP files in some way, such as via one of the following two different approaches:

```
 # Execute PHP when filenames end in .php
AddHandler application/x-httpd-php .php

# All files in this location are assumed to be PHP scripts.
```

6.4. Working with Large Numbers of Users

The trick to handling large numbers of users is to establish a clear, well-defined policy at the beginning and stick to it. It is essential to have the policy distributed to all users. Few of them will read it, but there isn't anything else you can do about it except be polite when they complain. With all the work we have done so far to secure dynamic request execution, some holes do remain. System accounts (virtual or not) can and will be used to attack your system or the neighboring accounts. A well-known approach to breaking into shared hosting web sites is through insecure configuration, working from another shared hosting account with the same provider.

Many web sites use PHP-based content management programs, but hosted on servers where PHP is configured to store session information in a single folder for all virtual accounts. Under such circumstances, it is probably trivial to hijack the program from a neighboring hosting account. If file permissions are not configured correctly and dynamic requests are executed as a single user, attackers can use PHP scripts to read other users' files and retrieve their data.

6.4.1. Web Shells

Though very few hosting providers give shells to their customers, few are aware that a shell is just a tool to make use of the access privileges customers already have. They do not need a shell to upload a web script to simulate a shell (such scripts are known as *web shells*), or even to upload a daemon and run it on the provider's server.

If you have not used a web shell before, you will be surprised how full-featured some of them are. For examples, see the following:

- CGITelnet.pl (http://www.rohitab.com/cgiscripts/cgitelnet.html)
- PhpShell (http://www.gimpster.com/wiki/PhpShell)
- PerlWebShell (http://yola.in-berlin.de/perlwebshell/)

You cannot stop users from running web shells, but by having proper filesystem configuration or virtual filesystems, you can make them a nonissue. Still, you may want to have cron scripts that look through customers' *cgi-bin/* folders searching for well-known web shells. Another possibility is to implement intrusion detection and monitor Apache output to detect traces of web shells in action.

6.4.2. Dangerous Binaries

When users are allowed to upload and execute their own binaries (and many are), that makes them potentially very dangerous. If the binaries are being executed safely (with an execution wrapper), the only danger comes from having a vulnerability in the operating system. This is where regular patching helps. As part of your operational procedures, be prepared to disable executable content upload, if a kernel vulnerability is discovered, until you have it patched.

Some people use their execution privileges to start daemons. (Or attackers exploit other people's execution privileges to do that.) For example, it is quite easy to upload and run something like Tiny Shell (http://www.cr0.net:8040/code/network/) on a high port on the machine. There are two things you can do about this:

- Monitor the execution of all user processes to detect the ones running for a long time. Such processes can be killed and reported. (However, ensure you do not kill the FastCGI processes.)
- Configure the firewall around the machine to only allow unsolicited traffic to a few required ports (80 and 443 in most cases) into the server, and not to allow any unrelated traffic out of the server. This will prevent the

Chapter 7. Access Control

Access control is an important part of security and is its most visible aspect, leading people to assume it is security. You may need to introduce access control to your system for a few reasons. The first and or most obvious reason is to allow some people to see (or do) what you want them to see/do while keeping the others out. However, you must also know who did what and when, so that they can be held accountable for their actions.

This chapter covers the following:

- Access control concepts
- HTTP authentication protocols
- Form-based authentication as an alternative to HTTP-based authentication
- Access control mechanisms built into Apache
- Single sign-on

7.1. Overview

Access control concerns itself with restricting access to authorized persons and with establishing accountability. There are four terms that are commonly used in discussions related to access control:

Identification

Process in which a user presents his identity

Authentication

Process of verifying the user is allowed to access the system

Authorization

Process of verifying the user is allowed to access a particular resource

Accountability

Ability to tell who accessed a resource and when, and whether the resource was modified as part of the access

From system users' point of view, they rarely encounter accountability, and the rest of the processes can appear to be a single step. When working as a system administrator, however, it is important to distinguish which operation is performed in which step and why. I have been very careful to word the definitions to reflect the true meanings of these terms.

Identification is the easiest process to describe. When required, users present their credentials so subsequent processes to establish their rights can begin. In real life, this is the equivalent of showing a pass upon entering a secure area.

The right of the user to access the system is established in the authentication step. This part of the process is often viewed as establishing someone's identity but, strictly speaking, this is not the case. Several types of information, called *factors*, are used to make the decision:

Something you know (Type 1)

This is the most commonly used authentication type. The user is required to demonstrate knowledge of some informatione.g., a password, passphrase, or PIN code.

Something you have (Type 2)

A Type 2 factor requires the user to demonstrate possession of some material access control element, usually a smart card or token of some kind. In a wider sense, this factor can include the time and location attributes of an access request, for example, "Access is allowed from the central office during normal work hours."

Something you are (Type 3)

Finally, a Type 3 factor treats the user as an access control element through the use of biometrics; that is, physical attributes of a user such as fingerprints, voiceprint, or eye patterns.

7.2. Authentication Methods

This section discusses three widely deployed authentication methods:

- Basic authentication
- Digest authentication
- Form-based authentication

The first two are built into the HTTP protocol and defined in RFC 2617, "HTTP Authentication: Basic and Digest Access Authentication" (http://www.ietf.org/rfc/rfc2617.txt). Form-based authentication is a way of moving the authentication problem from a web server to the application.

Other authentication methods exist (Windows NT challenge/response authentication and the Kerberos-based Negotiate protocol), but they are proprietary to Microsoft and of limited interest to Apache administrators.

7.2.1. Basic Authentication

Authentication methods built into HTTP use headers to send and receive authentication-related information. When a client attempts to access a protected resource the server responds with a *challenge*. The response is assigned a 401 HTTP status code, which means that authentication is required. (HTTP uses the word "authorization" in this context but ignore that for a moment.) In addition to the response code, the server sends a response header WWW-Authenticate, which includes information about the required authentication scheme and the authentication *realm*. The realm is a case-insensitive string that uniquely identifies (within the web site) the protected area. Here is an example of an attempt to access a protected resource and the response returned from the server:

```
 $ telnet www.apachesecurity.net 80
Trying 217.160.182.153...
Connected to www.apachesecurity.net.
Escape character is '^]'.
GET /review/ HTTP/1.0
Host: www.apachesecurity.net

HTTP/1.1 401 Authorization Required
Date: Thu, 09 Sep 2004 09:55:07 GMT
WWW-Authenticate: Basic realm="Book Review"
Connection: close
Content-Type: text/html
```

The first HTTP 401 response returned when a client attempts to access a protected resource is normally not displayed to the user. The browser reacts to such a response by displaying a pop-up window, asking the user to type in the login credentials. After the user enters her username and password, the original request is attempted again, this time with more information.

```
 $ telnet www.apachesecurity.net 80
Trying 217.160.182.153...
Connected to www.apachesecurity.net.
Escape character is '^]'.
GET /review/ HTTP/1.0
Host: www.apachesecurity.net
Authorization: Basic aXZhbnI6c2VjcmV0

HTTP/1.1 200 OK
Date: Thu, 09 Sep 2004 10:07:05 GMT
Connection: close
Content-Type: text/html
```

7.3. Access Control in Apache

Out of the box, Apache supports the Basic and Digest authentication protocols with a choice of plaintext or DBM files (documented in a later section) as backends. (Apache 2 also includes the *mod_auth_ldap* module, but it is considered experimental.) The way authentication is internally handled in Apache has changed dramatically in the 2.1 branch. (In the Apache 2 branch, odd-number releases are development versions. See http://cvs.apache.org/viewcvs.cgi/httpd-2.0/VERSIONING?view=markup for more information on new Apache versioning rules.) Many improvements are being made with little impact to the end users. For more information, take a look at the web site of the 2.1 Authentication Project at http://mod-auth.sourceforge.net.

Outside Apache, many third-party authentication modules enable authentication against LDAP, Kerberos, various database servers, and every other system known to man. If you have a special need, the Apache module repository at http://modules.apache.org is the first place to look.

7.3.1. Basic Authentication Using Plaintext Files

The easiest way to add authentication to Apache configuration is to use *mod_auth* , which is compiled in by default and provides Basic authentication using plaintext password files as authentication source.

You need to create a password file using the *htpasswd* utility (in the Apache */bin* folder after installation). You can keep it anywhere you want but ensure it is out of reach of other system users. I tend to keep the password file at the same place where I keep the Apache configuration so it is easier to find:

```
 # htpasswd -c /usr/local/apache/conf/auth.users ivanr
New password: ******
Re-type new password: ******
Adding password for user ivanr
```

This utility expects a path to a password file as its first parameter and the username as its second. The first invocation requires the -c switch, which instructs the utility to create a new password file if it does not exist. A look into the newly created file reveals a very simple structure:

```
 # cat /usr/local/apache/conf/auth.users
ivanr:EbsMlzzsDXiFg
```

You need the *htpasswd* utility to encrypt the passwords since storing passwords in plaintext is a bad idea. For all other operations, you can use your favorite text editor. In fact, you must use the text editor because *htpasswd* provides no features to rename accounts, and most versions do not support deletion of user accounts. (The Apache 2 version of the *httpasswd* utility does allow you to delete a user account with the -D switch.)

To password-protect a folder, add the following to your Apache configuration, replacing the folder, realm, and user file specifications with values relevant for your situation:

```
 <Directory /var/www/htdocs/review/>
    # Choose authentication protocol
    AuthType Basic
    # Define the security realm
    AuthName "Book Review"
    # Location of the user password file
    AuthUserFile /usr/local/apache/conf/auth.users
    # Valid users can access this folder and no one else
    Require valid-user
</Directory>
```

After you restart Apache, access to the folder will require valid login credentials.

7.3.1.1 Working with groups

Using one password file per security realm may work fine in simpler cases but does not work well when users are allowed access to some realms but not the others. Changing passwords for such users would require changes to all password files they belong to. A better approach is to have only one password file. The Require directive allows only

7.4. Single Sign-on

The term *single sign-on* (SSO) is used today to refer to several different problems, but it generally refers to a system where people can log in only once and have access to system-wide resources. What people mean when they say SSO depends on the context in which the term is used:

- SSO within a single organization
- SSO among many related organizations
- Internet-wide SSO among unrelated organizations

The term *identity management* is used to describe the SSO problem from the point of view of those who maintain the system. So what is the problem that makes implementing SSO difficult? Even within a single organization where the IT operations are under the control of a central authority, achieving all business goals by deploying a single system is impossible, no matter how complex the system. In real life, business goals are achieved with the use of many different components. For example, at minimum, every modern organization must enable their users to do the following:

- Log on to their workstations
- Send email (via an SMTP server)
- Read email (via a POP or IMAP server)

In most organizations, this may lead to users having three sets of unrelated credentials, so SSO is not achieved. And I haven't even started to enumerate all the possibilities. A typical organization will have many web applications (e.g., intranet, project management, content management) and many other network accounts (e.g., FTP servers). As the organization grows, the problem grows exponentially. Maintaining the user accounts and all the passwords becomes a nightmare for system administrators even if users simplify their lives by using a single password for all services. From the security point of view, a lack of central access control leads to complete failure to control access and to be aware of who is doing what with the services. On the other hand, unifying access to resources means that if someone's account is broken into, the attacker will get access to every resource available to the user. (In a non-SSO system, only one particular service would be compromised.) Imagine only one component that stores passwords insecurely on a local hard drive. Anyone with physical access to the workstation would be able to extract the password from the drive and use it to get access to other resources in the system.

SSO is usually implemented as a central database of user accounts and access privileges (usually one set of credentials per user used for all services). This is easier said than done since many of the components were not designed to play well with each other. In most cases, the SSO problem lies outside the realm of web server administration since many components are not web servers. Even in the web server space, there are many brands (Apache, Microsoft IIS, Java-based web servers) and SSO must work across all of them.

A decent SSO strategy is to use a Lightweight Directory Access Protocol (LDAP) server to store user accounts. Many web servers and other network servers support the use of LDAP for access control. Microsoft decided to use Kerberos (http://web.mit.edu/kerberos/www/) for SSO, but the problem with Kerberos is that all clients must be Kerberos-aware and most browsers still are not. In the Apache space, the *mod_auth_kerb* module (http://modauthkerb.sourceforge.net) can be configured to use Basic authentication to collect credentials from the user and check them against a Kerberos server, thus making Kerberos work with any browser.

Chapter 8. Logging and Monitoring

One of the most important tasks of an administrator is to configure a system to be secure, but it is also necessary to know it is secure. The only way to know a system is secure (and behaving correctly) is through informative and trustworthy log files. Though the security point of view is almost all we care about, we have other reasons to have good logs, such as to perform traffic analysis (which is useful for marketing) or to charge customers for the use of resources (billing and accounting).

Most administrators do not think about the logs much before an intrusion happens and only realize their configuration mistakes when it is discovered that critical forensic information is not available. In this chapter, we will cover the subjects of logging and monitoring, which are important to ensure the system records relevant information from a security perspective.

This chapter covers the following:

- Apache logging facilities
- Log manipulation
- Remote logging
- Logging strategies
- Log forensics
- Monitoring

8.1. Apache Logging Facilities

Apache can produce many types of logs. The two essential types are the access log, where all requests are noted, and the error log, which is designed to log various informational and debug messages, plus every exceptional event that occurs. Additional information can be found in module-specific logs, as is the case with *mod_ssl*, *mod_rewrite* and *mod_security*. The access log is created and written to by the module *mod_log_config*, which is not a part of the core, but this module is so important that everyone treats it as if it is.

8.1.1. Request Logging

You only need to be familiar with three configuration directives to manage request logging:

- LogFormat
- transferLog
- CustomLog

In fact, you will need to use only two. The CustomLog directive is so flexible and easy to use that you will rarely need to use transferLog in your configuration. (It will become clear why later.)

Other directives are available, but they are deprecated and should not be used because CustomLog can achieve all the necessary functionality. Some have been removed from Apache 2:

CookieLog

Deprecated, but still available

AgentLog

Deprecated and removed from Apache 2

RefererLog

Deprecated and removed from Apache 2

RefererIgnore

Deprecated and removed from Apache 2

8.1.1.1 LogFormat

Before covering the process of logging to files, consider the format of our log files. One of the benefits of Apache is its flexibility when it comes to log formatting. All this is owed to the LogFormat directive, whose default is the following, referred to as the Common Log Format (CLF):

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
```

The first parameter is a format string indicating the information to be included in a log file and the format in which it should be written; the second parameter gives the format string a name. You can decipher the log format using the

8.2. Log Manipulation

Apache does a good job with log format definition, but some features are missing, such as log rotation and log compression. Some reasons given for their absence are technical, and some are political:

- Apache usually starts as *root*, opens the log files, and proceeds to create child processes. Child processes inherit log file descriptors at birth; because of different permission settings, they would otherwise be unable to write to the logs. If Apache were to rotate the log files, it would have to create new file descriptors, and a mechanism would have to exist for children to "reopen" the logs.
- Some of the Apache developers believe that a web server should be designed to serve web pages, and should not concern itself with tasks such as log rotation.

Of course, nothing prevents third-party modules from implementing any kind of logging functionality, including rotation. After all, the default logging is done through a module (*mod_log_config*) without special privileges. However, at the time of this writing no modules exist that log to files and support rotation. There has been some work done on porting Cronolog (see Section 8.2.2.2 in the Section 8.2.2 section) to work as a module, but the beta version available on the web site has not been updated recently.

8.2.1. Piped Logging

Piped logging is a mechanism used to offload log manipulation from Apache and onto external programs. Instead of giving a configuration directive the name of the log file, you give it the name of a program that will handle logs in real time. A pipe character is used to specify this mode of operation:

```
CustomLog "|/usr/local/apache/bin/piped.pl /var/www/logs/piped_log" combined
```

All logging directives mentioned so far support piped logging. Many third-party modules also try to support this way of logging.

External programs used this way are started by the web server and restarted later if they die. They are started early, while Apache is still running as *root*, so they are running as *root*, too. Bugs in these programs can have significant security consequences. If you intend to experiment with piped logging, you will find the following proof-of-concept Perl program helpful to get you started:

```
#!/usr/bin/perl

use IO::Handle;

# check input parameters
if ((!@ARGV)||($#ARGV != 0)) {
    print "Usage: piped.pl <log filename>\n";
    exit;
}

# open the log file for appending, configuring
# autoflush to avoid potential data loss
$logfile = shift(@ARGV);
open(LOGFILE, ">>$logfile") || die "Failed to open $logfile for writing";
LOGFILE->autoflush(1);

# handle log entries until the end
while (my $logline = <STDIN>) {
    print LOGFILE $logline;
}

close(LOGFILE);
```

If you prefer C to Perl, every Apache distribution comes with C-based piped logging programs in the *support/* folder.

8.3. Remote Logging

Logging to the local filesystem on the same server is fine when it is the only server you have. Things get complicated as the number of servers rises. You may find yourself in one or more of the following situations:

- You have more than one server and want to have all your logs at one place.
- You have a cluster of web servers and must have your logs at one place.
- You want to increase system security by storing the logs safely to prevent intruders from erasing them.
- You want to have all event data centralized as part of a holistic system security approach.

The solution is usually to introduce a central logging host to the system, but there is no single ideal solution. I cover several approaches in the following sections.

8.3.1. Manual Centralization

The most natural way to centralize logs is to copy them across the network using the tools we already have, typically FTP, Secure File Transfer Program (SFTP), part of the Secure Shell package, or Secure Copy (SCP), also part of the SSH package. All three can be automated. As a bonus, SFTP and SCP are secure and allow us to transfer the logs safely across network boundaries.

This approach is nice, secure (assuming you do not use FTP), and simple to configure. Just add the transfer script to *cron*, allowing enough time for logs to be rotated. The drawback of this approach is that it needs manual configuration and maintenance and will not work if you want the logs placed on the central server in real time.

8.3.2. Syslog Logging

Logging via syslog is the default approach for most system administrators. The syslog protocol (see RFC 3164 at http://www.ietf.org/rfc/rfc3164.txt) is simple and has two basic purposes:

- Within a single host, messages are transmitted from applications to the syslog daemon via a domain socket.
- Between network hosts, syslog uses UDP as the transfer protocol.

Since all Unix systems come with *syslog* preinstalled, it is fairly easy to start using it for logging. A free utility, NTsyslog (http://ntsyslog.sourceforge.net), is available to enable logging from Windows machines.

The most common path a message will take starts with the application, through the local daemon, and across the network to the central logging host. Nothing prevents applications from sending UDP packets across the network directly, but it is often convenient to funnel everything to the localhost and decide what to do with log entries there, at a single location.

Apache supports syslog logging directly only for the error log. If the special keyword syslog is specified, all error messages will go to the syslog:

```
ErrorLog syslog:facility
```

The facility is an optional parameter, but you are likely to want to use it. Every syslog message consists of three parts: priority, facility, and the message. Priority can have one of the following eight values: debug, info, notice, warning

8.4. Logging Strategies

After covering the mechanics of logging in detail, one question remains: which strategy do we apply? That depends on your situation and no single perfect solution exists. Use Table 8-8 as a guideline.

Table 8-8. Logging strategy choices

Logging strategy	Situations when strategy is appropriate
Writing logs to the filesystem	• When there is only one machine or where each machine stands on its own. • If you are hosting static web sites and the web server is not viewed as a point of intrusion.
Database logging	• You have a need for ad hoc queries. If you are afraid the logging database might become a bottleneck (benchmark first), then put logs onto the filesystem first and periodically feed them to the database.
Syslog logging	• A syslog-based log centralization system is already in place.
Syslog logging with Syslog-NG (reliable, safe)	• Logs must be transferred across network boundaries and plaintext transport is not acceptable.
Manual centralization (SCP, SFTP)	• Logs must be transferred across network boundaries, but you cannot justify a full Syslog-NG system.
Spread toolkit	• You have a cluster of servers where there are several servers running the same site. • All other situations that involve more than one machine.

Here is some general advice about logging:

- Think about what you want from your logs and configure Apache accordingly.
- Decide how long you want to keep the logs. Decide at the beginning instead of keeping the logs forever or making up the rules as you go.
-

8.5. Log Analysis

Successful log analysis begins long before the need for it arises. It starts with the Apache installation, when you are deciding what to log and how. By the time something that requires log analysis happens, you should have the information to perform it.

If you are interested in log forensics, then Scan of the Month 31 (http://www.honeynet.org/scans/scan31/) is the web site you should visit. As an experiment, Ryan C. Barnett kept an Apache proxy open for a month and recorded every transaction in detail. It resulted in almost 300 MB of raw logs. The site includes several analyses of the abuse techniques seen in the logs.

A complete log analysis strategy consists of the following steps:

1. Ensure all Apache installations are configured to log sufficient information, prior to any incidents.
2. Determine all the log files where relevant information may be located. The access log and the error log are the obvious choices, but many other potential logs may contain useful information: the suEXEC log, the SSL log (it's in the error log on Apache 2), the audit log, and possibly application logs.
3. The access log is likely to be quite large. You should try to remove the irrelevant entries (e.g., requests for static files) from it to speed up processing. Watch carefully what is being removed; you do not want important information to get lost.
4. In the access log, try to group requests to sessions, either using the IP address or a session identifier if it appears in logs. Having the unique id token in the access log helps a lot since you can perform access log analysis much faster than you could with the full audit log produced by *mod_security*. The audit log is more suited for looking at individual requests.
5. Do not forget the attacker could be working from multiple IP addresses. Attackers often perform reconnaissance from one point but attack from another.

Log analysis is a long and tedious process. It involves looking at large quantities of data trying to make sense out of it. Traditional Unix tools (e.g., *grep*, *sed*, *awk*, and *sort*) and the command line are very good for text processing and, therefore, are a good choice for log file processing. But they can be difficult to use with web server logs because such logs contain a great deal of information. The bigger problem is that attackers often utilize evasion methods that must be taken into account during analysis, so a special tool is required. I have written one such tool for this book: *logscan*.

logscan parses log lines and allows field names to be used with regular expressions. For example, the following will examine the access log and list all requests whose status code is 500:

```
$ logscan access_log status 500
```

The parameters are the name of the log file, the field name, and the pattern to be used for comparison. By default, *logscan* understands the following field names, listed in the order in which they appear in access log entries:

- remote_addr
-

8.6. Monitoring

The key to running a successful project is to be in control. System information must be regularly collected for historical and statistical purposes and allow real-time notification when something goes wrong.

8.6.1. File Integrity

One of the system security best practices demands that every machine makes use of an integrity checker, such as Tripwire, to monitor file integrity. The purpose of an integrity checker is to detect an intruder early, so you can act quickly and contain the intrusion.

As a special case, integrity checkers can be applied against the user files in the web server tree. I believe Tripwire was among the first to offer such a product, in the form of an Apache module. The product was discontinued, and the problem was probably due to the frequent changes that take place on most web sites. Of what use is a security measure that triggers the alarm daily? Besides, many web sites construct pages dynamically, with the content stored in databases, so the files on disk are not that relevant any more. Still, in a few cases where reputation is extremely important (e.g., for governments), this approach has some merit.

8.6.2. Event Monitoring

The first thing to consider when it comes to event monitoring is whether to implement real-time monitoring. Real-time monitoring sounds fancy, but unless an effort is made to turn it into a useful tool, it can do more harm than good. Imagine the following scenario:

A new application is being deployed. The web server uses *mod_security* to detect application-level attacks. Each time an attack is detected, the request is denied with status code 403 (forbidden), and an email message is sent to the developers. Excited, developers read every email in the beginning. After a while, with no time to verify each attack, all developers have message filters that move such notifications into a separate folder, and no one looks at them any more.

This is real-time monitoring gone bad. Real problems often go undetected because of too many false positives. A similar lesson can be learned from the next example, too:

Developers have installed a script to check the operation of the application every five minutes. When a failure is detected, the script sends an email, which generates a series of mobile phone messages to notify all team members. After some time in operation, the system breaks in the middle of the night. Up until the problem was resolved two hours later (by the developer who was on duty at that time), all five members of the development team received 25 phone messages each. Since many turned off their phones a half an hour after the problem was first detected (because they could not sleep), some subsequent problems that night went undetected.

The two cases I have just described are not something I invented to prove a point. There are numerous administrative and development teams suffering like that. These problems can be resolved by following four rules:

Funnel all events into log files

Avoid using ad-hoc notification mechanisms (application emails, scripts triggered by ErrorDocument, module actions). Instead, send all events to the error log, implement some mechanism to watch that one location, and act when necessary.

Implement notification only when necessary

Do not send notifications about attacks you have blocked. Notifications should serve to inform others about real problems. A good example of a required real-time notification is an SQL query failure. Such an event is a sign of a badly written application or an attacker practicing SQL injection. Either way, it must be addressed immediately.

Chapter 9. Infrastructure

In this chapter, we take a step back from a single Apache server to discuss the infrastructure and the architecture of the system as a whole. Topics include:

- Application isolation strategies
- Host security
- Network security
- Use of a reverse proxy, including use of web application firewalls
- Network design

We want to make each element of the infrastructure as secure as it can be and design it to work securely as if the others did not exist. We must do the following:

- Do everything to keep attackers out.
- Design the system to minimize the damage of break in.
- Detect compromises as they occur.

Some sections of this chapter (the ones on host security and network security) discuss issues that not only relate to Apache, but also could be applied to running any service. I will mention them briefly so you know you need to take care of them. If you wish to explore these other issues, I recommend of the following books:

- Practical Unix & Internet Security by Simson Garfinkel, Gene Spafford, and Alan Schwartz (O'Reilly)
- Internet Site Security by Erik Schetina, Ken Green, and Jacob Carlson (Addison-Wesley)
- Linux Server Security by Michael D. Bauer (O'Reilly)
- Network Security Hacks by Andrew Lockhart (O'Reilly)

Network Security Hacks is particularly useful because it is concise and allows you to find an answer quickly. If you need to do something, you look up the hack in the table of contents, and a couple of pages later you have the problem solved.

9.1. Application Isolation Strategies

Choosing a correct application isolation strategy can have a significant effect on a project's security. Ideally, a strategy will be selected early in the project's life, as a joint decision of the administration and the development team. Delaying the decision may result in the inability to deploy certain configurations.

9.1.1. Isolating Applications from Servers

Your goal should be to keep each application separated from the operating system it resides on. It is simple to do when deploying the application and will help in the future. The following rules of thumb apply:

- Store the web application into a single folder on disk. An application that occupies a single folder is easy to back up, move to another server, or install onto a freshly installed server. When disaster strikes, you will need to act quickly and you do not want anything slowing you down.
- If the application requires a complex installation (for example, third-party Apache modules or specific PHP configuration), treat Apache and its modules as part of the application. This will make the application easy to move from one server to another.
- Keep the application-specific configuration data close to the application, referencing such data from the main configuration file (*httpd.conf*) using the Include directive.

In addition to facilitating disaster recovery, another reason to keep an application isolated is to guard servers from intrusions that take place through applications. Such isolation contains the intrusion and makes the life of the attacker more difficult due to the absence of the tools he would like to use to progress further. This kind of isolation is done through the chroot process (see Chapter 2).

9.1.2. Isolating Application Modules

Isolating application modules from each other helps reduce damage caused by a break-in. The idea is not to put all your eggs into one basket. First, you need to determine whether there is room for isolation. When separating the application into individual logical modules, you need to determine whether there are modules that are accessed by only one class of user. Each module should be separated from the rest of the application to have its own:

- Domain name
- IP address
- System user account
- Database access account
- Accounts for access to other resources (e.g., LDAP)

This configuration will allow for maximal security and maximal configuration flexibility. If you cannot accommodate such separation initially, due to budget constraints, you should plan for it anyway and upgrade the system when the opportunity arises.

9.2. Host Security

Going backward from applications, host security is the first layer we encounter. Though we will continue to build additional defenses, the host must be secured as if no additional protection existed. (This is a recurring theme in this book.)

9.2.1. Restricting and Securing User Access

After the operating system installation, you will discover many shell accounts active in the */etc/passwd* file. For example, each database engine comes with its own user account. Few of these accounts are needed. Review every active account and cancel the shell access of each account not needed for server operation. To do this, replace the shell specified for the user in */etc/password* with */bin/false*. Here is a replacement example:

```
ivanr:x:506:506::/home/users/ivanr:/bin/bash
```

with:

```
ivanr:x:506:506::/home/users/ivanr:/bin/false
```

Restrict whom you provide shell access. Users who are not security conscious represent a threat. Work to provide some other way for them to do their jobs without the shell access. Most users only need to have a way to transport files and are quite happy using FTP for that. (Unfortunately, FTP sends credentials in plaintext, making it easy to break in.)

Finally, secure the entry point for interactive access by disabling insecure plaintext protocols such as Telnet, leaving only secure shell (SSH) as a means for host access. Configure SSH to refuse direct *root* logins, by setting PermitRootLogin to no in the *sshd_config* file. Otherwise, in an environment where the *root* password is shared among many administrators, you may not be able to tell who was logged on at a specific time.

If possible, do not allow users to use a mixture of plaintext (insecure) and encrypted (secure) services. For example, in the case of the FTP protocol, deploy *Secure FTP* (SFTP) where possible. If you absolutely must use a plaintext protocol and some of the users have shells, consider opening two accounts for each such user: one account for use with secure services and the other for use with insecure services. Interactive logging should be forbidden for the latter; that way a compromise of the account is less likely to lead to an attacker gaining a shell on the system.

9.2.2. Deploying Minimal Services

Every open port on a host represents an entry point for an attacker. Closing as many ports as possible increases the security of a host. Operating systems often have many services enabled by default. Use the *netstat* tool on the command line to retrieve a complete listing of active TCP and UDP ports on the server:

```
# netstat -nlp
                                                                    PID/
Proto Recv-Q Send-Q Local Address   Foreign Address   State       Program name
tcp        0      0 0.0.0.0:3306    0.0.0.0:*         LISTEN      963/mysqld
tcp        0      0 0.0.0.0:110     0.0.0.0:*         LISTEN      834/xinetd
tcp        0      0 0.0.0.0:143     0.0.0.0:*         LISTEN      834/xinetd
tcp        0      0 0.0.0.0:80      0.0.0.0:*         LISTEN      13566/httpd
tcp        0      0 0.0.0.0:21      0.0.0.0:*         LISTEN      1060/proftpd
tcp        0      0 0.0.0.0:22      0.0.0.0:*         LISTEN      -
tcp        0      0 0.0.0.0:23      0.0.0.0:*         LISTEN      834/xinetd
tcp        0      0 0.0.0.0:25      0.0.0.0:*         LISTEN      979/sendmail
udp        0      0 0.0.0.0:514     0.0.0.0:*                     650/syslogd
```

Now that you know which services are running, turn off the ones you do not need. (You will probably want port 22 open so you can continue to access the server.) Turning services off permanently is a two-step process. First you need to turn the running instance off:

```
# /etc/init.d/proftpd stop
```

Then you need to stop the service from starting the next time the server boots. The procedure depends on the

9.3. Network Security

Another step backward from host security and we encounter network security. We will consider the network design a little bit later. For the moment, I will discuss issues that need to be considered in this context:

- Firewall usage
- Centralized logging
- Network monitoring
- External monitoring

A central firewall is mandatory. The remaining three steps are highly recommended but not strictly necessary.

9.3.1. Firewall Usage

Having a central firewall in front, to guard the installation, is a mandatory requirement. In most cases, the firewalling capabilities of the router will be used. A dedicated firewall can be used where very high-security operation is required. This can be a brand-name solution or a Unix box.

The purpose of the firewall is to enforce the site-access policy, making public services public and private services private. It also serves as additional protection for misconfigured host services. Most people think of a firewall as a tool that restricts traffic coming from the outside, but it can (and should) also be used to restrict traffic that is originating from inside the network.

If you have chosen to isolate application modules, having a separate IP address for each module will allow you to control access to modules directly on the firewall.

Do not depend only on the firewall for protection. It is only part of the overall protection strategy. Being tough on the outside does not work if you are weak on the inside; once the perimeter is breached the attacker will have no problems breaching internal servers.

9.3.2. Centralized Logging

As the number of servers grows, the ability to manually follow what is happening on each individual server decreases. The "standard" growth path for most administrators is to use host-based monitoring tools or scripts and use email messages to be notified of unusual events. If you follow this path, you will soon discover you are getting too many emails and you still don't know what is happening and where.

Implementing a centralized logging system is one of the steps toward a solution for this problem. Having the logs at one location ensures you are seeing everything. As an additional benefit, centralization enhances the overall security of the system: if a single host on the network is breached the attacker may attempt to modify the logs to hide her tracks. This is more difficult when logs are duplicated on a central log server. Here are my recommendations:

- Implement a central log server on a dedicated system by forwarding logs from individual servers.
- Keep (and rotate) a copy of the logs on individual servers to serve as backup.
-

9.4. Using a Reverse Proxy

A *proxy* is an intermediary communication device. The term "proxy" commonly refers to a *forward proxy*, which is a gateway device that fetches web traffic on behalf of client devices. We are more interested in the opposite type of proxy. *Reverse proxies* are gateway devices that isolate servers from the Web and accept traffic on their behalf.

There are two reasons to add a reverse proxy to the network: security and performance. The benefits coming from reverse proxies stem from the concept of centralization: by having a single point of entry for the HTTP traffic, we are increasing our monitoring and controlling capabilities. Therefore, the larger the network, the more benefits we will have. Here are the advantages:

Unified access control

Since all requests come in through the proxy, it is easy to see and control them all. Also known as a central point of policy enforcement.

Unified logging

Similar to the previous point, we need to collect logs only from one device instead of devising complex schemes to collect logs from all devices in the network.

Improved performance

Transparent caching, content compression, and SSL termination are easy to implement at the reverse proxy level.

Application isolation

With a reverse proxy in place, it becomes possible (and easy) to examine every HTTP request and response. The proxy becomes a sort of umbrella, which can protect vulnerable web applications.

Host and web server isolation

Your internal network may consist of many different web servers, some of which may be legacy systems that cannot be replaced or fixed when broken. Preventing direct contact with the clients allows the system to remain operational and safe.

Hiding of network topology

The more attackers know about the internal network, the easier it is to break in. The topology is often exposed through a carelessly managed DNS. If a network is guarded by a reverse proxy system, the outside world need not know anything about the internal network. Through the use of private DNS servers and private address space, the network topology can be hidden.

There are some disadvantages as well:

Increased complexity

Adding a reverse proxy requires careful thought and increased effort in system maintenance.

9.5. Network Design

A well-designed network is the basis for all other security efforts. Though we are dealing with Apache security here, our main subject alone is insufficient. Your goal is to implement a switched, modular network where services of different risk are isolated into different network segments.

Figure 9-1 illustrates a classic demilitarized zone (DMZ) network architecture.

Figure 9-1. Classic DMZ architecture

This architecture assumes you have a collection of backend servers to protect and also assumes danger comes from one direction only, which is the Internet. A third zone, DMZ, is created to work as an intermediary between the danger outside and the assets inside.

Ideally, each service should be isolated onto its own server. When circumstances make this impossible (e.g., financial reasons), try not to combine services of different risk levels. For example, combining a public email server with an internal web server is a bad idea. If a service is not meant to be used directly from the outside, moving it to a separate server would allow you to move the service out of the DMZ and into the internal LAN.

For complex installations, it may be justifiable to create classes of users. For example, a typical business system will operate with:

- Public users
- Partners (extranet)
- Internal users (intranet)

With proper planning, each of these user classes can have its own DMZ, and each DMZ will have different privileges with regards to access to the internal LAN. Multiple DMZs allow different classes of users to access the system via different means. To participate in high-risk systems, partners may be required to access the network via a virtual private network (VPN).

To continue to refine the network design, there are four paths from here:

Network hardening

General network-hardening elements can be introduced into the network to make it more secure. They include things such as dedicated firewalls, a central logging server, intrusion detection systems, etc.

Chapter 10. Web Application Security

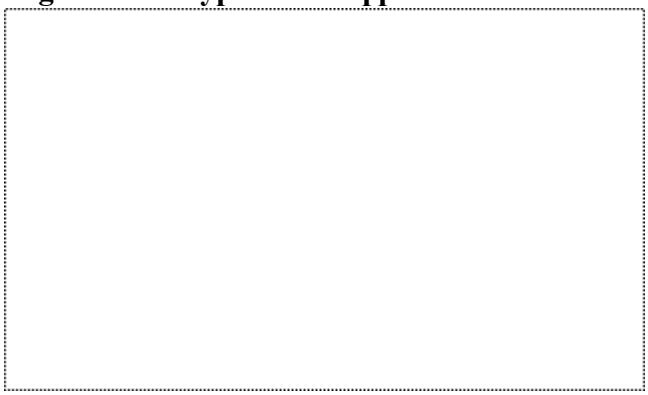
This chapter covers web application security on a level that is appropriate for the profile of this book. That's not an easy task: I've tried to adequately but succinctly cover all relevant points, without delving into programming too much.

To compensate for the lack of detail in some spots, I have provided a large collection of web application security links. In many cases the links point to security papers that were the first to introduce the problem, thereby expanding the web application security book of knowledge.

Unless you are a programmer, you will not need to concern yourself with every possible detail presented in this chapter. The idea is to grasp the main concepts and to be able to spot major flaws at a first glance. As is typical with the 20/80 rule: invest 20 percent of your effort to get 80 percent of the desired results.

The reason web application security is difficult is because a web application typically consists of many very different components glued together. A typical web application architecture is illustrated in Figure 10-1. In this figure, I have marked the locations where some frequent flaws and attacks occur.

Figure 10-1. Typical web application architecture



To build secure applications developers must be well acquainted with individual components. In today's world, where everything needs to be completed yesterday, security is often an afterthought. Other factors have contributed to the problem as well:

- HTTP was originally designed for document exchange, but it evolved into an application deployment platform. Furthermore, HTTP is now used to transport whole new protocols (e.g., SOAP). Using one port to transport multiple protocols significantly reduces the ability of classic firewall architectures to control what traffic is allowed; it is only possible to either allow or deny everything that goes over a port.
- The Web grew into a mandatory business tool. To remain competitive, companies must deploy web applications to interact with their customers and partners.
- Being a plaintext protocol, HTTP does not require any special tools to perform exploitation. Most attacks can be performed manually, using a browser or a telnet client. In addition, many attacks are very easy to execute.

Security issues should be addressed at the beginning of web application development and throughout the development lifecycle. Every development team should have a security specialist on board. The specialist should be the one to educate other team members, spread awareness, and ensure there are no security lapses. Unfortunately this is often not possible in real life.

If you are a system administrator, you may be faced with a challenge to deploy and maintain systems of unknown

10.1. Session Management Attacks

HTTP is a stateless protocol. It was never designed to handle sessions. Though this helped the Web take off, it presents a major problem for web application designers. No one anticipated the Web being used as an application platform. It would have been much better to have session management built right into the HTTP standard. But since it wasn't, it is now re-implemented by every application separately. Cookies were designed to help with sessions but they fall short of finishing the job.

10.1.1. Cookies

Cookies are a mechanism for web servers and web applications to remember some information about a client. Prior to their invention, there was no way to uniquely identify a client. The only other piece of information that can be used for identification is the IP address. Workstations on local networks often have static, routable IP addresses that rarely change. These addresses can be used for pretty reliable user tracking. But in most other situations, there are too many unknowns to use IP addresses for identification:

- Sometimes workstations are configured to retrieve an unused IP address from a pool of addresses at boot time, usually using a DHCP server. If users turn off their computers daily, their IP addresses can (in theory) be different each day. Thus, an IP address used by one workstation one day can be assigned to a different workstation the next day.
- Some workstations are not allowed to access web content directly and instead must do so through a web proxy (typically as a matter of corporate policy). The IP address of the proxy is all that is visible from the outside.
- Some workstations think they are accessing the Web directly, but their traffic is being changed in real time by a device known as a *Network Address Translator* (NAT). The address of the NAT is all that is visible from the outside.
- Dial-up users and many DSL users regularly get assigned a different IP address every time they connect to the Internet. Only a small percentage of dial-up users have their own IP addresses.
- Some dial-up users (for example, those coming through AOL) can have a different IP address on each HTTP request, as their providers route their original requests through a cluster of transparent HTTP proxies.
- Finally, some users do not want their IP addresses to be known. They configure their clients to use so-called open proxies and route HTTP requests through them. It is even possible to chain many proxies together and route requests through all of them at once.
- Even in the case of a computer with a permanent real (routable) IP address, many users could be using the same workstation. User tracking via an IP address would, therefore, view all these users as a single user.

Something had to be done to identify users. With stateful protocols, you at least know the address of the client throughout the session. To solve the problem for stateless protocols, people at Netscape invented cookies. Perhaps Netscape engineers thought about fortune cookies when they thought of the name. Here is how they work:

1. Upon first visit (first HTTP request), the site stores information identifying a session into a cookie and sends the cookie to the browser

10.2. Attacks on Clients

Though attacks on clients are largely irrelevant for web application security (the exception being the use of JavaScript to steal session tokens), we will cover them briefly from the point of view that if you are in charge of a web application deployment, you must cover all attack vectors.

10.2.1. Typical Client Attack Targets

Here are some of the things that may be targeted:

- Browser flaws
- Java applets
- Browser plug-ins (such as Flash or Shockwave)
- JavaScript/VBScript embedded code

Attacking any of these is difficult. Most of the early flaws have been corrected. Someone may attempt to create a custom Mozilla plug-in or Internet Explorer ActiveX component, but succeeding with that requires the victim to willingly accept running the component. If your users are doing that, then you have a bigger problem with all the viruses spreading around. The same users can easily become victims of phishing (see the next section).

Internet Explorer is a frequent target because of its poor security record. In my opinion, Internet Explorer, Outlook, and Outlook Express should not be used in environments that require a high level of security until their security improves. You are better off using software such as Mozilla Suite (or now separate packages Firefox and Thunderbird).

10.2.2. Phishing

Phishing is a shorter version of the term *password fishing*. It is used for attacks that try to trick users into submitting passwords and other sensitive private information to the attacker by posing as someone else. The process goes like this:

1. Someone makes a copy of a popular password-protected web site (we are assuming passwords are protecting something of value). Popular Internet sites such as Citibank, PayPal, and eBay are frequent targets.
2. This person sends forged email messages to thousands, or even millions, of users, pretending the messages are sent from the original web site and directing people to log in to the forged site. Attackers usually use various techniques to hide the real URL the users are visiting.
3. Naïve users will attempt to login and the attacker will record their usernames and passwords. The attacker can now redirect the user to the real site. The user, thinking there was a glitch, attempts to log in again (this time to the real site), succeeds, thinks everything is fine, and doesn't even notice the credentials were stolen.
4. The attacker can now access the original password-protected area and exploit this power, for example by transferring funds from the victim's account to his own.

10.3. Application Logic Flaws

Application logic flaws are the result of a lack of understanding of the web application programming model. Programmers are often deceived when something looks right and they believe it works right too. Most flaws can be tracked down to two basic errors:

- Information that comes from the client is trusted and no (or little) validation is performed.
- Process state is not maintained on the server (in the application).

I explain the errors and the flaws resulting from them through a series of examples.

10.3.1. Cookies and Hidden Fields

Information stored in cookies and hidden form fields is not visible to the naked eye. However, it can be accessed easily by viewing the web page source (in the case of hidden fields) or configuring the browser to display cookies as they arrive. Browsers in general do not allow anyone to change this information, but it can be done with proper tools. (Paros, described in the Appendix A, is one such tool.)

Because browsers do not allow anyone to change cookie information, some programmers use cookies to store sensitive information (application data). They send cookies to the client, accept them back, and then use the application data from the cookie in the application. However, the data has already been *tainted*.

Imagine an application that uses cookies to authenticate user sessions. Upon successful authentication, the application sends the following cookie to the client (I have emphasized the application data):

```
Set-Cookie: authenticated=true; path=/; domain=www.example.com
```

The application assumes that whoever has a cookie named authenticated containing true is an authenticated user. With such a concept of security, the attacker only needs to forge a cookie with the same content and access the application without knowing the username or the password.

It is a similar story with hidden fields. When there is a need in the application to perform a two-step process, programmers will often perform half of the processing in the first step, display step one results to the user in a page, and transmit some internal data into the second step using hidden fields. Though browsers provide no means for users to change the hidden fields, specialized tools can. The correct approach is to use the early steps only to collect and validate data and then repeat validation and perform the main task in the final step.

Allowing users to interfere with application internal data often results in attackers being able to do the following:

- Change product price (usually found in simpler shopping carts)
- Gain administrative privileges (*vertical privilege escalation*)
- Impersonate other users (*horizontal privilege escalation*)

An example of this type of flaw can be found in numerous form-to-email scripts. To enable web designers to have data sent to email without a need to do any programming, all data is stored as hidden form fields:

```
<form action="/cgi-bin/FormMail" method="POST">
<input type="hidden" name="subject" value="Call me back">
<input type="hidden" name="recipient" value="sales@example.com">
<!-- the visible part of the form follows here -->
</form>
```

10.4. Information Disclosure

The more bad guys know about your system, the easier it becomes to find a way to compromise it. Information disclosure refers to the family of flaws that reveal inside information.

10.4.1. HTML Source Code

There is more in HTML pages than most people see. A thorough analysis of HTML page source code can reveal useful information. The structure of the source code is itself important because it can tell a lot about the person who wrote it. You can judge that person's design and programming skills and learn what to expect.

HTML comments

You can commonly find comments in HTML code. For web designers, it is the only place for comments other designers can see. Even programmers, who should be writing comments in code and not in HTML (comments in code are never sent to browsers) sometimes make a mistake and put in information that should not be there.

JavaScript code

The JavaScript code can reveal even more about the coder's personality. Parts of the code that deal with data validation can reveal information about application business rules. Programmers sometimes fail to implement data validation on the server side, relying on the client-side JavaScript instead. Knowing the business rules makes it easier to test for boundary cases.

Tool comments and metadata

Tools used to create pages often put comments in the code. Sometimes they reveal paths on the filesystem. You can identify the tool used, which may lead to other discoveries (see the "Predictable File Locations" section below).

10.4.2. Directory Listings

A directory listing is a dynamically generated page showing the contents of a requested folder. Web servers creating such listings are only trying to be helpful, and they usually do so only after realizing the default index file (*index.html*, *index.php*, etc.) is absent. Directory listings are sometimes served to the client even when a default index file exists, as a result of web server vulnerability. This happens to be one of the most frequent Apache problems, as you can see from the following list of releases and their directory listing vulnerabilities. (The Common Vulnerability and Exposure numbers are inside the parentheses; see http://cve.mitre.org.)

- v1.3.12 Requests can cause directory listing on NT (CVE-2000-0505).
- v1.3.17 Requests can cause directory listing to be displayed (CVE-2001-0925).
- v1.3.20 Multiviews can cause a directory listing to be displayed (CVE-2001-0731).
- v1.3.20 Requests can cause directory listing to be displayed on Win32 (CVE-2001-0729).

A directory-listing service is not needed in most cases and should be turned off. Having a web server configured to produce directory listings where they are not required should be treated as a configuration error.

10.5. File Disclosure

File disclosure refers to the case when someone manages to download a file that would otherwise remain hidden or require special authorization.

10.5.1. Path Traversal

Path traversal occurs when directory backreferences are used in a path to gain access to the parent folder of a subfolder. If the software running on a server fails to resolve backreferences, it may also fail to detect an attempt to access files stored outside the web server tree. This flaw is known as *path traversal* or *directory traversal*. It can exist in a web server (though most web servers have fixed these problems) or in application code. Programmers often make this mistake.

If it is a web server flaw, an attacker only needs to ask for a file she knows is there:

```
http://www.example.com/../../etc/passwd
```

Even when she doesn't know where the document root is, she can simply increase the number of backreferences until she finds it.

Apache 1 will always respond with a 404 response code to any request that contains a URL-encoded slash (%2F) in the filename even when the specified file exists on the filesystem. Apache 2 allows this behavior to be configured at runtime using the AllowEncodedSlashes directive.

10.5.2. Application Download Flaws

Under ideal circumstances, files will be downloaded directly using the web server. But when a nontrivial authorization scheme is needed, the download takes place through a script after the authorization. Such scripts are web application security hot spots. Failure to validate input in such a script can result in arbitrary file disclosure.

Imagine a set of pages that implement a download center. Download happens through a script called *download.php*, which accepts the name of the file to be downloaded in a parameter called filename. A careless programmer may form the name of the file by appending the filename to the base directory:

```
$file_path = $repository_path + "/" + $filename;
```

An attacker can use the path traversal attack to request any file on the web server:

```
http://www.example.com/download.php?filename=../../etc/passwd
```

You can see how I have applied the same principle as before, when I showed attacking the web server directly. A naïve programmer will not bother with the repository path, and will accept a full file path in the parameter, as in:

```
http://www.example.com/download.php?filename=/etc/passwd
```

A file can also be disclosed to an attacker through a vulnerable script that uses a request parameter in an include statement:

```
include($file_path);
```

PHP will attempt to run the code (making this flaw more dangerous, as I will discuss later in the section "Code Execution"), but if there is no PHP code in the file it will output the contents of the file to the browser.

10.5.3. Source Code Disclosure

Source code disclosure usually happens when a web server is tricked into displaying a script instead of executing it. A popular way of doing this is to modify the URL enough to confuse the web server (and prevent it from determining

10.6. Injection Flaws

Finally, we reach a type of flaw that can cause serious damage. If you thought the flaws we have covered were mostly harmless you would be right. But those flaws were a preparation (in this book, and in successful compromise attempts) for what follows.

Injection flaws get their name because when they are used, malicious user-supplied data flows through the application, crosses system boundaries, and gets injected into another system component. System boundaries can be tricky because a text string that is harmless for PHP can turn into a dangerous weapon when it reaches a database.

Injection flaws come in as many flavors as there are component types. Three flaws are particularly important because practically every web application can be affected:

SQL injection

When an injection flaw causes user input to modify an SQL query in a way that was not intended by the application author

Cross-site scripting (XSS)

When an attacker gains control of a user browser by injecting HTML and Java-Script code into the page

Operating system command execution

When an attacker executes shell commands on the server

Other types of injection are also feasible. Papers covering LDAP injection and XPath injection are listed in the section Section 10.9.

10.6.1. SQL Injection

SQL injection attacks are among the most common because nearly every web application uses a database to store and retrieve data. Injections are possible because applications typically use simple string concatenation to construct SQL queries, but fail to sanitize input data.

10.6.1.1 A working example

SQL injections are fun if you are not at the receiving end. We will use a complete programming example and examine how these attacks take place. We will use PHP and MySQL 4.x. You can download the code from the book web site, so do not type it.

Create a database with two tables and a few rows of data. The database represents an imaginary bank where my wife and I keep our money.

```
 CREATE DATABASE sql_injection_test;

USE sql_injection_test;

CREATE TABLE customers (
    customerid INTEGER NOT NULL,
    username CHAR(32) NOT NULL,
    password CHAR(32) NOT NULL,
    PRIMARY KEY(customerid)
);

INSERT INTO customers ( customerid, username, password )
```

10.7. Buffer Overflows

Buffer overflow occurs when an attempt is made to use a limited-length buffer to store a larger piece of data. Because of the lack of boundary checking, some amount of data will be written to memory locations immediately following the buffer. When an attacker manipulates program input, supplying specially crafted data payload, buffer overflows can be used to gain control of the application.

Buffer overflows affect C-based languages. Since most web applications are scripted (or written in Java, which is not vulnerable to buffer overflows), they are seldom affected by buffer overflows. Still, a typical web deployment can contain many components written in C:

- Web servers, such as Apache
- Custom Apache modules
- Application engines, such as PHP
- Custom PHP modules
- CGI scripts written in C
- External systems

Note that external systems such as databases, mail servers, directory servers and other servers are also often programmed in C. That the application itself is scripted is irrelevant. If data crosses system boundaries to reach the external system, an attacker could exploit a vulnerability.

A detailed explanation of how buffer overflows work falls outside the scope of this book. Consult the following resources to learn more:

- The Shellcoder's Handbook: Discovering and Exploiting Security Holes by Jack Koziol et al. (Wiley)
- "Practical Code Auditing" by Lurene A. Grenier (http://www.daemonkitty.net/lurene/papers/Audit.pdf)
- "Buffer Overflows Demystified" by Murat Balaban (http://www.enderunix.org/docs/eng/bof-eng.txt)
- "Smashing The Stack For Fun And Profit" by Aleph One (http://www.insecure.org/stf/smashstack.txt)
- "Advanced Doug Lea's malloc exploits" by jp@corest.com (http://www.phrack.org/phrack/61/p61-0x06_Advanced_malloc_exploits.txt)
- "Taking advantage of nonterminated adjacent memory spaces" by twitch@vicar.org (http://www.phrack.org/phrack/56/p56-0x0e)

10.8. Evasion Techniques

Intrusion detection systems (IDSs) are an integral part of web application security. In Chapter 9, I introduced web application firewalls (also covered in Chapter 12), whose purpose is to detect and reject malicious requests.

Most web application firewalls are signature-based. This means they monitor HTTP traffic looking for signature matches, where this type of "signature" is a pattern that suggests an attack. When a request is matched against a signature, an action is taken (as specified by the configuration). But if an attacker modifies the attack payload in some way to have the same meaning for the target but not to resemble a signature the web application firewall is looking for, the request will go through. Techniques of attack payload modification to avoid detection are called *evasion techniques*.

Evasion techniques are a well-known tool in the TCP/IP-world, having been used against network-level IDS tools for years. In the web security world, evasion is somewhat new. Here are some papers on the subject:

- "A look at whisker's anti-IDS tactics" by Rain Forest Puppy (http://www.apachesecurity.net/archive/whiskerids.html)
- "IDS Evasion Techniques and Tactics" by Kevin Timm (http://www.securityfocus.com/printable/infocus/1577)

10.8.1. Simple Evasion Techniques

We start with the simple yet effective evasion techniques:

Using mixed case characters

This technique can be useful for attackers when attacking platforms (e.g., Windows) where filenames are not case sensitive; otherwise, it is useless. Its usefulness rises, however, if the target Apache includes *mod_speling* as one of its modules. This module tries to find a matching file on disk, ignoring case and allowing up to one spelling mistake.

Character escaping

Sometimes people do not realize you can escape any character by preceding the character with a backslash character (\), and if the character does not have a special meaning, the escaped character will convert into itself. Thus, \d converts to d. It is not much but it is enough to fool an IDS. For example, an IDS looking for the pattern id would not detect a string i\d, which has essentially the same meaning.

Using whitespace

Using excessive whitespace, especially the less frequently thought of characters such as TAB and new line, can be an evasion technique. For example, if an attacker creates an SQL injection attempt using DELETE FROM (with two spaces in between the words instead of one), the attack will be undetected by an IDS looking for DELETE FROM (with just one space in between).

10.8.2. Path Obfuscation

Many evasion techniques are used in attacks against the filesystem. For example, many methods can obfuscate paths to make them less detectable:

Self-referencing directories

10.9. Web Application Security Resources

Web security is not easy because it requires knowledge of many different systems and technologies. The resources listed here are only a tip of the iceberg.

10.9.1. General Resources

- HTTP: The Definitive Guide by David Gourley and Brian Totty (O'Reilly)
- RFC 2616, "Hypertext Transfer Protocol HTTP/1.1" (http://www.ietf.org/rfc/rfc2616.txt)
- HTML 4.01 Specification (http://www.w3.org/TR/html401/)
- JavaScript Central (http://devedge.netscape.com/central/javascript/)
- ECMAScript Language Specification (http://www.ecma-international.org/publica-tions/files/ecma-st/ECMA-262.pdf)
- ECMAScript Components Specification (http://www.ecma-international.org/pub-lications/files/ecma-st/ECMA-290.pdf)

For anyone wanting to seriously explore web security, a fair knowledge of components (e.g., database systems) making up web applications is also necessary.

10.9.2. Web Application Security Resources

Web application security is a young discipline. Few books cover the subject in depth. Researchers everywhere, including individuals and company employees, regularly publish papers that show old problems in new light.

- Hacking Exposed: Web Applications by Joel Scambray and Mike Shema (McGraw-Hill/Osborne)
- Hack Notes: Web Security Portable Reference by Mike Shema (McGraw-Hill/Osborne)
- PHP Security by Chris Shiflett (O'Reilly)
- Open Web Application Security Project (http://www.owasp.org)
- "Guide to Building Secure Web Applications" by OWASP (Open Web Application Security Project) (http://www.owasp.org/documentation/guide.html)
- SecurityFocus Web Application Security Mailing List (webappsec@securityfocus.com) (http://www.securityfocus.com/archive/107)

Chapter 11. Web Security Assessment

The purpose of a web system security assessment is to determine how tight security is. Many deployments get it wrong because the responsibility to ensure a web system's security is split between administrators and developers. I have seen this many times. Neither party understands the whole system, yet they have responsibility to ensure security.

The way I see it, web security is the responsibility of the system administrator. With the responsibility assigned to one party, the job becomes an order of magnitude easier. If you are a system administrator, think about it this way:

It is your server. That makes you responsible!

To get the job done, you will have to approach the other side, web application development, and understand how it is done. The purpose of Chapter 10 was to give you a solid introduction to web application security issues. The good news is that web security is very interesting! Furthermore, you will not be expected to create secure code, only judge it.

The assessment methodology laid down in this chapter is what I like to call "lightweight web security assessment methodology." The word "lightweight" is there because the methodology does not cover every detail, especially the programming parts. In an ideal world, web application security should only be assessed by web application security professionals. They need to concern themselves with programming details. I will assume you are not this person, you have many tasks to do, and you do not do web security full time. Have the 20/80 rule in mind: expend 20 percent of the effort to get 80 percent of the benefits.

Though web security professionals can benefit from this book, such professionals will, however, use the book as a starting point and make that 80 percent of additional effort that is expected of them. A complete web security assessment consists of three complementary parts. They should be executed in the following order:

Black-box testing

Testing from the outside, with no knowledge of the system.

White-box testing

Testing from the inside, with full knowledge of the system.

Gray-box testing

Testing that combines the previous two types of testing. Gray-box testing can reflect the situation that might occur when an attacker can obtain the source code for an application (it could have been leaked or is publicly available). In such circumstances, the attacker is likely to set up a copy of the application on a development server and practice attacks there.

Before you continue, look at the Appendix A, where you will find a list of web security tools. Knowing how something works under the covers is important, but testing everything manually takes away too much of your precious time.

11.1. Black-Box Testing

In black-box testing, you pretend you are an outsider, and you try to break in. This useful technique simulates the real world. The less you know about the system you are about to investigate, the better. I assume you are doing black-box assessment because you fall into one of these categories:

- You want to increase the security of your own system.
- You are helping someone else secure their system.
- You are performing web security assessment professionally.

Unless you belong to the first category, you must ensure you have permission to perform black-box testing. Black-box testing can be treated as hostile and often illegal. If you are doing a favor for a friend, get written permission from someone who has the authority to provide it.

Ask yourself these questions: Who am I pretending to be? Or, what is the starting point of my assessment? The answer depends on the nature of the system you are testing. Here are some choices:

- A member of the general public
- A business partner of the target organization
- A customer on the same shared server where the target application resides
- A malicious employee
- A fellow system administrator

Different starting points require different approaches. A system administrator may have access to the most important servers, but such servers are (hopefully) out of reach of a member of the public. The best way to conduct an assessment is to start with no special privileges and examine what the system looks like from that point of view. Then continue upward, assuming other roles. While doing all this, remember you are doing a web security assessment, which is a small fraction of the subject of information security. Do not cover too much territory, or you will never finish. In your initial assessment, you should focus on the issues mostly under your responsibility.

As you perform the assessment, record everything, and create an information trail. If you know something about the infrastructure beforehand, you must prove you did not use it as part of black-box testing. You can use that knowledge later, as part of white-box testing.

Black-box testing consists of the following steps:

1. Information gathering (passive and active)
2. Web server analysis
3.

11.2. White-Box Testing

White-box testing is the complete opposite of what we have been doing. The goal of black-box testing was to rely only on your own resources and remain anonymous and unnoticed; here we can access anything anywhere (or so the theory goes).

The key to a successful white-box review is having direct contact and cooperation from developers and people in charge of system maintenance. Software documentation may be nonexistent, so you will need help from these people to understand the environment to the level required for the assessment.

To begin the review, you need the following:

- Complete application documentation and the source code.
- Direct access to application developers and system administrators. There is no need for them to be with you all the time; having their telephone numbers combined with a meeting or two will be sufficient.
- Unrestricted access to the production server or to an exact system replica. You will need a working system to perform tests since looking at the code is not enough.

The process of white-box testing consists of the following steps:

1. Architecture review
2. Configuration review
3. Functional review

At the end of your white-box testing, you should have a review report that documents your methodology, contains review notes, lists notices, warnings, and errors, and offers recommendations for improvement.

11.2.1. Architecture Review

The purpose of the architecture review is to pave the way for the actions ahead. A good understanding of the application is essential for a successful review. You should examine the following:

Application security policy

If you are lucky, the application review will begin with a well-defined security policy in hand. If such a thing does not exist (which is common), you will have difficulties defining what "security" means. Where possible, a subproject should be branched out to create the application security policy. Unless you know what needs to be protected, it will not be possible to determine whether the system is secure enough. If a subproject is not a possibility, you will have to sketch a security policy using common sense. This security policy will suffer from being focused too much on technology, and based on your assumptions about the business (which may be incorrect). In any case, you will definitely need something to guide you through the rest of the review.

Application modules

11.3. Gray-Box Testing

In the third and final phase of security assessment, the black-box testing procedures are executed again but this time using the knowledge acquired in the white-box testing phase. This is similar to the type of testing an attacker might do when he has access to the source code, but here you have a slight advantage because you know the layout of the files on disk, the configuration, and changes made to the original source code (if any). This time you are also allowed to have access to the target system while you are testing it from the outside. For example, you can look at the application logs to discover why some of your attacks are failing.

The gray-box testing phase is the time to confirm or deny the assumptions about vulnerabilities you made in the black-box phase. For example, maybe you thought Apache was vulnerable to a particular problem but you did not want to try to exploit it at that time. Looking at it from the inside, it is much easier and quicker to determine if your assumption was correct.

Chapter 12. Web Intrusion Detection

In spite of all your efforts to secure a web server, there is one part you do not and usually cannot control in its entirety: web applications. Web application design, programming, and maintenance require a different skill set. Even if you have the skills, in a typical organization these tasks are usually assigned to someone other than a system administrator. But the problem of ensuring adequate security remains. This final chapter suggests ways to secure applications by treating them as black boxes and examining the way they interact with the environment. The techniques that do this are known under the name intrusion detection.

This chapter covers the following:

- Evolution of intrusion detection
- Basic intrusion detection principles
- Web application firewalls
- *mod_security*

12.1. Evolution of Web Intrusion Detection

Intrusion detection has been in use for many years. Its purpose is to detect attacks by looking at the network traffic or by looking at operating system events. The term *intrusion prevention* is used to refer to systems that are also capable of preventing attacks.

Today, when people mention intrusion detection, in most cases they are referring to a *network intrusion detection system* (NIDS). An NIDS works on the TCP/IP level and is used to detect attacks against any network service, including the web server. The job of such systems, the most popular and most widely deployed of all IDSs, is to monitor raw network packets to spot malicious payload. *Host-based intrusion detection systems* (HIDSs), on the other hand, work on the host level. Though they can analyze network traffic (only the traffic that arrives to that single host), this task is usually left to NIDSs. Host-based intrusion is mostly concerned with the events that take place on the host (such as users logging in and out and executing commands) and the system error messages that are generated. An HIDS can be as simple as a script watching a log file for error messages, as mentioned in Chapter 8. Integrity validation programs (such as Tripwire) are a form of HIDS. Some systems can be complex: one form of HIDS uses system call monitoring on a kernel level to detect processes that behave suspiciously.

Using a single approach for intrusion detection is insufficient. *Security information management* (SIM) systems are designed to manage various security-relevant events they receive from *agents*, where an agent can listen to the network traffic or operating system events or can work to obtain any other security-relevant information.

Because many NIDSs are in place, a large effort was made to make the most of them and to use them for web intrusion detection, too. Though NIDSs work well for the problems they were designed to address and they can provide some help with web intrusion detection, they do not and cannot live up to the full web intrusion detection potential for the following reasons:

- NIDSs were designed to work with TCP/IP. The Web is based around the HTTP protocol, which is a completely new vocabulary. It comes with its own set of problems and challenges, which are different from the ones of TCP/IP.
- The real problem is that web applications are not simple users of the HTTP protocol. Instead, HTTP is only used to carry the application-specific data. It is as though each application builds its own protocol on top of HTTP.
- Many new protocols are deployed on top of HTTP (think of Web Services, XML-RPC, and SOAP), pushing the level of complexity further up.
- Other problems, such as the inability of an NIDS to see through encrypted SSL channels (which most web applications that are meant to be secure use) and the inability to cope with a large amount of web traffic, make NIDSs insufficient tools for web intrusion detection.

Vendors of NIDSs have responded to the challenges by adding extensions to better understand HTTP. The term *deep-inspection firewalls* refers to systems that make an additional effort to understand the network traffic on a higher level. Ultimately, a new breed of IDSs was born. *Web application firewalls* (WAFs), also known as *web application gateways*, are designed specifically to guard web applications. Designed from the ground up to support HTTP and to exploit its transactional nature, web application firewalls often work as reverse proxies. Instead of going directly to the web application, a request is rerouted to go to a WAF first and only allowed to proceed if deemed safe.

Web application firewalls were designed from the ground up to deal with web attacks and are better suited for that purpose. NIDSs are better suited for monitoring on the network level and cannot be replaced for that purpose.

12.2. Using mod_security

mod_security is a web application firewall module I developed for the Apache web server. It is available under the open source GPL license, with commercial support and commercial licensing as an option. I originally designed it as a means to obtain a proper audit log, but it grew to include other security features. There are two versions of the module, one for each major Apache branch, and they are almost identical in functionality. In the Apache 2 version, *mod_security* uses the advanced filtering API available in that version, making interception of the response body possible. The Apache 2 version is also more efficient in terms of memory consumption. In short, *mod_security* does the following:

- Intercepts HTTP requests before they are fully processed by the web server
- Intercepts the request body (e.g., the POST payload)
- Intercepts, stores, and optionally validates uploaded files
- Performs anti-evasion actions automatically
- Performs request analysis by processing a set of rules defined in the configuration
- Intercepts HTTP responses before they are sent back to the client (Apache 2 only)
- Performs response analysis by processing a set of rules defined in the configuration
- Takes one of the predefined actions or executes an external script when a request or a response fails analysis (a process called *detection*)
- Depending on the configuration, a failed request may be prevented from being processed, and a failed response may be prevented from being seen by the client (a process called *prevention*)
- Performs audit logging

In this section, I present a deployment guide for *mod_security*, but the principles behind it are the same and can be applied to any web application firewall. For a detailed reference manual, visit the project documentation area at http://www.modsecurity.org/documentation/.

12.2.1. Introduction

The basic ingredients of every *mod_security* configuration are:

- Anti-evasion features
- Encoding validation features

Appendix A. Tools

When I was young, I had a lot of fun playing a game called Neuromancer, which takes place in a world created by William Gibson, in the book with the same name. The game was very good at giving a similar feeling (I now know) to that of a hacker learning about and making his way through a system for the first time. The Internet was young at the time (1989), but the game had it all: email, newsgroups, servers, hacking, and artificial intelligence. (I am still waiting for that last one to appear in real life.) I was already interested in programming at that time, but I think the game pushed me somewhat toward computer security.

In the game, your success revolved around having the right tools at the right time. It did not allow you to create your own tools, so the action was mostly in persuading shady individuals to give, trade, or sell tools. In real life, these tools would be known under the name *exploits*. (It was acceptable to use them in the game because the player was fighting the evil AI.) Now, many years later, it is funny to realize that real life is much more interesting and creative than any game will ever be. Still, the security business feels much the same as in that game I played ages ago. For both, it is important to do the following:

- Start with a solid understanding of the technology
- Have and use the correct tools
- Write your own tools

This appendix contains a list of tools you may find useful to perform the activities mentioned throughout the book. While some of these are not essential (meaning there are lower-level tools that would get the work done), they are great time-savers.

A.1. Learning Environments

The best way to learn about web application security is to practice development and assessment. This may prove difficult as not everyone has a web application full of vulnerabilities lying around. (Assessing someone else's application without her consent is unacceptable.) The answer is to use a controlled environment in which programming mistakes have been planted on purpose.

Two such environments are available:

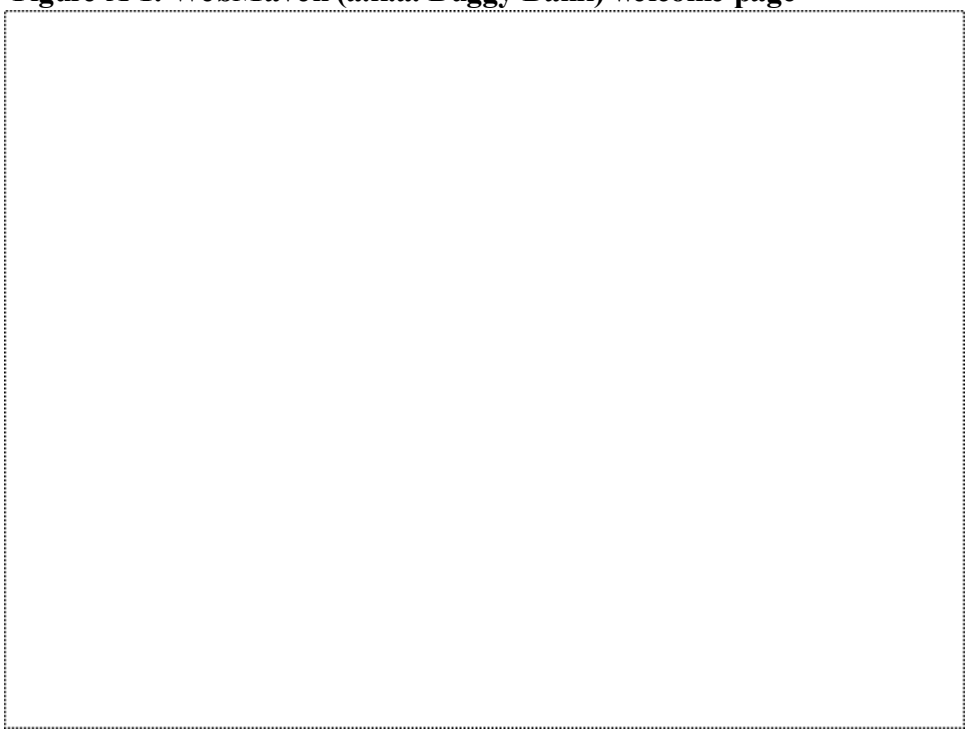
- WebMaven (http://www.mavensecurity.com/webmaven/)
- WebGoat (http://www.owasp.org/software/webgoat.html)

A.1.1. WebMaven

WebMaven is a simple interactive learning environment for web application security. It was originally developed by David Rhoades from Maven Security and subsequently released as open source. Written in Perl, the application is easy to install on Unix and Windows computers.

WebMaven simulates an online banking system ("Buggy Bank"), which offers customers the ability to log in, log out, view account status, and transfer funds. As you can imagine, the application contains many (ten, according to the user manual) intentional errors. Your task is to find them. If you get stuck, you can find the list of vulnerabilities at the end of the user manual. Looking at the vulnerability list defeats the purpose of the learning environment so I strongly encourage you to try it on your own for as long as you can. You can see the welcome page of the Buggy Bank in Figure A-1.

Figure A-1. WebMaven (a.k.a. Buggy Bank) welcome page



A.1.2. WebGoat

WebGoat (Figure A-2) is a Java-based web security environment for learning. The installation script is supposed to install Tomcat if it is not already installed, but as of this writing, it doesn't work. (It attempts to download an older version of Tomcat that is not available for download any more.) You should install Tomcat manually first.

A.2. Information-Gathering Tools

On Unix systems, most information gathering tools are available straight from the command line. It is the same on Windows, provided Cygwin (http://www.cygwin.com) is installed.

A.2.1. Online Tools at TechnicalInfo

If all you have is a browser, TechnicalInfo contains a set of links (http://www.technicalinfo.net/tools/) to various information-gathering tools hosted elsewhere. Using them can be cumbersome and slow, but they get the job done.

A.2.2. Netcraft

Netcraft (http://www.netcraft.co.uk) is famous for its "What is that site running?" service, which identifies web servers using the Server header. (This is not completely reliable since some sites hide or change this information, but many sites do not.) Netcraft is interesting not because it tells you which web server is running at the site, but because it keeps historical information around. In some cases, this information can reveal the real identity of the web server.

This is exactly what happened with the web server hosting my web site www.modsecurity.org. I changed the web server signature some time ago, but the old signature still shows in Netcraft results.

Figure A-3 reveals another problem with changing server signatures. It lists my server as running Linux and Internet Information Server simultaneously, which is implausible. In this case, I am using the signature "Microsoft-IIS/5.0" as a bit of fun. If I were to use it seriously, I would need to pay more attention to what signature I was choosing.

Figure A-3. Historical server information from Netcraft

A.2.3. Sam Spade

Sam Spade (http://www.samspade.org/ssw/), a freeware network query tool from Steve Atkins will probably provide you with all the network tools you need if your desktop is running Windows. Sam Spade includes all the passive tools you would expect, plus some advanced features on top of those:

- Simple multiaddress port scanning.

A.3. Network-Level Tools

You will need a range of network-level tools for your day-to-day activities. These command-line tools are designed to monitor and analyze traffic or allow you to create new traffic (e.g., HTTP requests).

A.3.1. Netcat

Using a simple Telnet client will work well for most manually executed HTTP requests but it pays off to learn the syntax of Netcat. Netcat is a TCP and UDP client and server combined in a single binary, designed to be scriptable and used from a command line.

Netcat is available in two versions:

- @stake Netcat (the original, http://www.securityfocus.com/tools/137)
- GNU Netcat (http://netcat.sourceforge.net/)

To use it as a port scanner, invoke it with the -z switch (to initiate a scan) and -v to tell it to report its findings:

```
$ nc -v -z www.modsecurity.org 1-1023
Warning: inverse host lookup failed for 217.160.182.153:
         Host name lookup failure
www.modsecurity.org [217.160.182.153] 995 (pop3s) open
www.modsecurity.org [217.160.182.153] 993 (imaps) open
www.modsecurity.org [217.160.182.153] 443 (https) open
www.modsecurity.org [217.160.182.153] 143 (imap) open
www.modsecurity.org [217.160.182.153] 110 (pop3) open
www.modsecurity.org [217.160.182.153] 80 (http) open
www.modsecurity.org [217.160.182.153] 53 (domain) open
www.modsecurity.org [217.160.182.153] 25 (smtp) open
www.modsecurity.org [217.160.182.153] 23 (telnet) open
www.modsecurity.org [217.160.182.153] 22 (ssh) open
www.modsecurity.org [217.160.182.153] 21 (ftp) open
```

To create a TCP server on port 8080 (as specified by the -p switch), use the -l switch:

```
$ nc -l -p 8080
```

To create a TCP proxy, forwarding requests from port 8080 to port 80, type the following. (We need the additional pipe to take care of the flow of data back from the web server.)

```
$ mknod ncpipe p
$ nc -l -p 8080 < ncpipe | nc localhost 80 > ncpipe
```

A.3.2. Stunnel

Stunnel (http://www.stunnel.org) is a universal SSL driver. It can wrap any TCP connection into an SSL channel. This is handy when you want to use your existing, non-SSL tools, to connect to an SSL-enabled server. If you are using Stunnel Versions 3.x and older, all parameters can be specified on the command line. Here is an example:

```
$ stunnel -c -d 8080 -r www.amazon.com:443
```

By default, Stunnel stays permanently active in the background. This command line tells Stunnel to go into client mode (-c), listen locally on port 8080 (-d) and connect to the remote server www.amazon.com on port 443 (-r). You can now use any plaintext tool to connect to the SSL server through Stunnel running on port 8080. I will use telnet and perform a HEAD request to ensure it works:

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to debian.
Escape character is '^]'.
HEAD / HTTP/1.0
```

A.4. Web Security Scanners

Similar to how network security scanners operate, web security scanners try to analyze publicly available web resources and draw conclusions from the responses.

Web security scanners have a more difficult job to do. Traditional network security revolves around publicly known vulnerabilities in well-known applications providing services (it is rare to have custom applications on the TCP level). Though there are many off-the-shelf web applications in use, most web applications (or at least the interesting ones) are written for specific purposes, typically by in-house teams.

A.4.1. Nikto

Nikto (http://www.cirt.net/code/nikto.shtml) is a free web security scanner. It is an open source tool available under the GPL license. There is no support for GUI operation, but the command-line options work on Unix and Windows systems. Nikto focuses on three web-related issues:

- Web server misconfiguration
- Default files and scripts (which are sometimes insecure)
- Outdated software
- Known vulnerabilities

Nikto cannot be aware of vulnerabilities in custom applications, so you will have to look for them yourself. Looking at how it is built and what features it supports, Nikto is very interesting:

- Written in Perl, uses libwhisker
- Supports HTTP and HTTPS
- Comes with a built-in signature database, showing patterns that suggest attacks; this database can be automatically updated
- Allows the use of a custom signature database
- Supports Perl-based plug-ins
- Supports TXT, HTML, or CVS output

If Perl is your cup of tea you will find Nikto very useful. With some knowledge of libwhisker, and the internal workings of Nikto, you should be able to automate the boring parts of web security assessment by writing custom plug-ins.

Nikto's greatest weakness is that it relies on the pre-built signature database to be effective. As is often the case with open source projects, this database does not seem to be frequently updated.

A.5. Web Application Security Tools

Web security tools provide four types of functionality, and there is a growing trend to integrate all the types into a single package. The four different types are:

Scanners

Execute a predetermined set of requests, analyzing responses to detect configuration errors and known vulnerabilities. They can discover vulnerabilities in custom applications by mutating request parameters.

Crawlers

Map the web site and analyze the source code of every response to discover "invisible" information: links, email addresses, comments, hidden form fields, etc.

Assessment proxies

Standing in the middle, between a browser and the target, assessment proxies record the information that passes by, and allow requests to be modified on the fly.

Utilities

Utilities used for brute-force password attacks, DoS attacks, encoding and decoding of data.

Many free (and some open source) web security tools are available:

- Paros (http://www.parosproxy.org)
- Burp proxy (http://www.portswigger.net/proxy/)
- Brutus (password cracker; http://www.hoobie.net/brutus/)
- Burp spider (http://portswigger.net/spider/)
- Sock (http://portswigger.net/sock/)
- WebScarab (http://www.owasp.org/software/webscarab.html)

These tools are rich in functionality but lacking in documentation and quality control. Some functions in their user interfaces can be less than obvious (this is not to say commercial tools are always user friendly), so expect to spend some time figuring out how they work. The trend is to use Java on the client side, making the tools work on most desktop platforms.

Paros and WebScarab compete for the title of the most useful and complete free tool. The Burp tools show potential, but lack integration and polish.

A.6. HTTP Programming Libraries

When all else fails, you may have to resort to programming to perform a request or a series of requests that would be impossible otherwise. If you are familiar with shell scripting, then the combination of *expect* (a tool that can control interactive programs programmatically), *netcat*, *curl*, and *stunnel* may work well for you. (If you do not already have *expect* installed, download it from http://expect.nist.gov.)

For those of you who are more programming-oriented, turning to one of the available HTTP programming libraries will allow you to do what you need fast:

libwww-perl (http://lwp.linpro.no/lwp/)

A collection of Perl modules that provide the functionality needed to programmatically generate HTTP traffic.

libcurl (http://curl.haxx.se/libcurl/)

The core library used to implement curl. Bindings for 23 languages are available.

libwhisker (http://www.wiretrip.net/rfp/lw.asp)

A Perl library that automates many HTTP-related tasks. It even supports some IDS evasion techniques transparently. A SecurityFocus article on libwhisker, "Using Libwhisker" by Neil Desai (http://www.securityfocus.com/infocus/1798), provides useful information on the subject.

Jakarta Commons HttpClient (http://jakarta.apache.org/commons/httpclient/)

If you are a Java fan, you will want to go pure Java, and you can with HttpClient. Feature-wise, the library is very complete. Unfortunately, every release comes with an incompatible programming interface.

Colophon

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

The animal on the cover of Apache Security is an Arabian horse (Equus caballus). Thousands of years ago, Bedouin tribes of the Arabian Peninsula (now comprising Syria, Iraq, and Iran) began breeding these horses as war mounts. Desert conditions were harsh, so Arabian horses lived in close proximity to their owners, sometimes even sharing their tents. This breed, known for its endurance, speed, intelligence, and close affinity to humans, evolved and flourished in near isolation before gaining popularity throughout the rest of the world.

The widespread enjoyment of Arabians as pleasure horses and endurance racers is generally attributed to the strict breeding of the Bedouins. According to the Islamic people, the Arabian horse was a gift from Allah. Its broad forehead, curved profile, wide-set eyes, arched neck, and high tail are distinct features of the Arabian breed, and these characteristics were highly valued and obsessed over during the breeding process. Because the Bedouins valued purity of strain above all else, many tribes owned only one primary strain of horse. These strains, or families, were named according to the tribe that bred them, and the genealogy of strains was always traced through the dam. Mythical stories accompanied any recitation of a substrain's genealogy. The daughters and granddaughters of legendary mares were much sought after by powerful rulers. One such case occurred around the 14th century, when Sultan Nacer Mohamed Ibn Kalaoun paid well over the equivalent of $5.5 million for a single mare.

Many Arabian pedigrees can still be traced to desert breeding. The Bedouins kept no written breeding records, but since they placed such high value on purity, the designation "desert-bred" is accepted as an authentic verification of pure blood. Arabians are also commonly crossed with other breeds, including thoroughbreds, Morgans, paint horses, Appaloosas, and quarter horses. Today, Arabian horses continue to be distinguished by their bloodlines. Breeding them involves a constant crossing of strains.

Matt Hutchinson was the production editor for Apache Security . GEX, Inc. provided production services. Darren Kelly, Lydia Onofrei, Claire Cloutier, and Emily Quill provided quality control.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is an original engraving from the 19th century. Emma Colby produced the cover layout with Adobe InDesign CS using Adobe's ITC Garamond font.

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

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

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index