MODSECURITY HANDBOOK

The Complete Guide to the Popular
Open Source Web Application Firewall
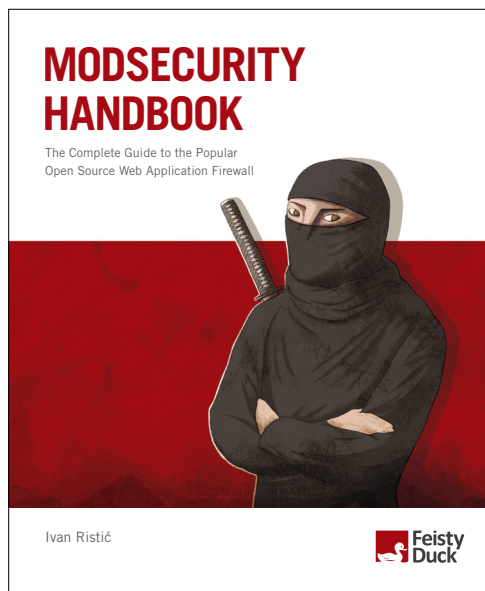
Free edition: Getting Started

Ivan Ristić

Feisty Duck

Everything you need to know about ModSecurity, in one place

MODSECURITY HANDBOOK

The Complete Guide to the Popular
Open Source Web Application Firewall

Ivan Ristić

Feisty Duck

ModSecurity Handbook

Ivan Ristić

Feisty Duck

LONDON

ModSecurity Handbook: Getting Started

by Ivan Ristić

Copyright © 2010-2013 Feisty Duck Limited. All rights reserved.

ISBN: 978-1-907117-06-0

Development version (revision 654).

First published in March 2010. Fully revised in April 2012.

Feisty Duck Limited
www.feistyduck.com
contact@feistyduck.com

Address:
6 Acantha Court
Montpelier Road
London W5 2QP
United Kingdom

Production editor: Jelena Girić-Ristić

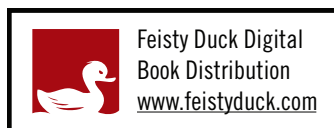**Copyeditor:** Nancy Kotary

Cover design: Peter Jovanović

Cover illustration: Maja Veselinović

Technical reviewer: Brian Rectanus

Feisty Duck Digital
Book Distribution
www.feistyduck.com

Table of Contents

Preface to the Free Edition

Welcome to *ModSecurity Handbook: Getting Started*. This is a special free version that consists of the first 4 chapters of the full book. Since *ModSecurity Handbook* went into print—slightly over 2 years ago—I realized that there were many new users of ModSecurity who were not yet interested in mastering this tool (and wouldn't consider buying a book), but could benefit from having access to better documentation. This version offers exactly that, with an introduction to ModSecurity in Chapter 1, and coverage of installation and configuration in Chapters 2, 3, and 4. Enjoy.

Preface

I didn't mean to write this book, I really didn't. Several months ago I started to work on the second edition of *Apache Security*, deciding to rewrite the ModSecurity chapter first. A funny thing happened: the ModSecurity chapter kept growing and growing. It hit 40 pages. It hit 80 pages. And then I realized that I was nowhere near the end. That was all the excuse I needed to put *Apache Security* aside—for the time being—and focus on a ModSecurity book instead.

I admit that I couldn't be happier, although it was an entirely emotional decision. After spending years working on ModSecurity, I knew it had so much more to offer, yet the documentation wasn't there to show the way. But it is now, I am thrilled to say. The package is complete: you have an open source tool that is able to compete with the best commercial products out there, *and* you have the documentation to match.

With this book I am also trying something completely new—*continuous writing and publishing*. You see, I had published my first book with a major publisher, but I never quite liked the process. It was too slow. You write a book pretty much in isolation, you publish it, and then you never get to update it. I was never happy with that, and that's why I decided to do things differently this time.

Simply said, *ModSecurity Handbook* is a living book. Every time I make a change, a new digital version is made available to you. If I improve the book based on your feedback, you get the improvements as soon as I make them. If you prefer a paper book, you can still get it of course, through the usual channels. Although I can't do anything about updating the paper version of the book, we can narrow the gap slightly by pushing out book updates even between editions. That means that, even when you get the paper version (as most people seem to prefer to), it is never going to be too much behind the digital version.

Scope and Audience

This book exists to document every single aspect of ModSecurity and to teach you how to use it. It is as simple as that. ModSecurity is a fantastic tool, but it is let down by the poor quality of the documentation. As a result, the adoption is not as good as it could be; application security is difficult on its own and you don't really want to struggle with poorly documented tools

too. I felt a responsibility to write this book and show how ModSecurity can compete with commercial web application firewalls, in spite of being the underdog. Now that the book is finished, I feel I've done a proper job with ModSecurity.

If you are interested in application security, you are my target audience. Even if you're not interested in application security as such, and only want to deal with your particular problems (it's difficult to find a web application these days that's without security problems), you are still my target audience.

You don't need to know anything about ModSecurity to get started. If you just follow the book from the beginning, you will find that every new chapter advances a notch. Even if you are a long-time ModSecurity user, I believe you will benefit from a fresh start. I will let you in on a secret—I have. There's nothing better for completing one's knowledge than having to write about a particular topic. I suspect that long-time ModSecurity users will especially like the second half of the book, which discusses many advanced topics and often covers substantial new ground.

But, there is only so much a book can cover. *ModSecurity Handbook* assumes you already know how to operate the Apache web server. You don't have to be an expert, but you do need to know how to install, configure, and run it. If you don't know how to do that already, you should get my first book, *Apache Security*. I wrote it five years ago, but it's still remarkably fresh. (Ironically, it is only the ModSecurity chapter in *Apache Security* that is completely obsolete. But that's why you have this book.)

On the other end, *ModSecurity Handbook* will teach you how to use ModSecurity and write good rules, but it won't teach you application security. In my earlier book, *Apache Security*, I included a chapter that served as an introduction to application security, but, even then, I was barely able to mention all that I wanted, and the chapter was still the longest chapter in the book. Since then, the application security field has exploded and now you have to read several books and dozens of research papers just to begin to understand it.

Contents

Once you go past the first chapter, which is the introduction to the world of ModSecurity, the rest of the book consists of roughly three parts. In the first part, you learn how to install and configure ModSecurity. In the second part, you learn how to write rules. As for the third part, you could say that it contains the advanced stuff—a series of chapters each dedicated to one important aspect of ModSecurity.

At the end of the book is the official reference documentation, reproduced with the permission from Breach Security.

Chapter 1, *Introduction*, is the foundation of the book. It contains a gentle introduction to ModSecurity, and then explains what it can and cannot do. The main usage scenarios are listed to help you identify where you can use ModSecurity in your environment. The middle of the

chapter goes under the hood of ModSecurity to give you an insight into how it works, and finishes with an overview of the key areas you will need to learn in order to deploy it. The end of the chapter lists a series of resources (sites, mailing lists, tools, etc.) that you will find useful in your day-to-day work.

Chapter 2, *Installation*, teaches you how to install ModSecurity, either compiling from source (using one of the released versions or downloading straight from the development repository), or by using one of the available binary packages, on Unix and Windows alike.

Chapter 3, *Configuration*, explains how each of the available configuration directives should be used. By the end of the chapter, you get a complete overview of the configuration options and will have a solid default configuration for all your ModSecurity installations.

Chapter 4, *Logging*, deals with the logging features of ModSecurity. The two main logging facilities explained are the debug log, which is useful in rule writing, and the audit log, which is used to log complete transaction data. Special attention is given to remote logging, which you'll need to manage multiple sensors, or to use any of the user-friendly tools for alert management. File interception and validation is covered in detail. The chapter ends with an advanced section of logging, which explains how to selectively log traffic, and how to use the sanitation feature to prevent sensitive data from being stored in the logs.

Chapter 5, *Rule Language Overview*, is the first of the three chapters that deal with rule writing. This chapter contains an overview of the entire rule language, which will get you started as well as give you a feature map to which you can return whenever you need to deal with a new problem.

Chapter 6, *Rule Language Tutorial*, teaches how to write rules, and how to write them well. It's a very fun chapter that adopts a gradual approach, introducing the features one by one. By the end of the chapter, you will know everything about writing individual rules.

Chapter 7, *Rule Configuration*, completes the topic of rule writing. It takes a step back to view the rules as the basic block for policy building. You first learn how to put a few rules together and add them to the configuration, as well as how the rules interact with Apache's ability to use different configuration contexts for different sites and different locations within sites. The chapter spends a great deal of time making sure you take advantage of the inheritance feature, which helps make ModSecurity configuration much easier to maintain.

Chapter 8, *Persistent Storage*, is quite possibly the most exciting chapter in the book. It describes the persistent storage mechanism, which enables you to track data and events over time and thus opens up an entire new dimension of ModSecurity. This chapter is also the most practical one in the entire book. It gives you the rules for periodic alerting, brute force attack detection, denial of service attack detection, session and user management, fixing session management weaknesses, and more.

Chapter 9, *Practical Rule Writing*, is, as the name suggests, a tour through many of the practical activities you will perform in your day-to-day work. The chapter starts by covering whitelisting, virtual patching, IP address reputation and blacklisting. You then learn how to integrate with other Apache modules, with practical examples that show how to perform conditional logging and fix insecure session cookies. Special attention is given to the topic of blocking; several approaches, starting from the simple to the very sophisticated, are presented. A section on regular expressions gets you up to speed with the most important ModSecurity operator. The chapter ends with a discussion of rule sets, discussing how to use the rule sets others have written, as well as how to write your own.

Chapter 10, *Performance*, covers several performance-related topics. It opens with an overview of where ModSecurity usually spends its time, a list of common configuration mistakes that should be avoided, and a list of approaches that result in better performance. The second part of the chapter describes how to monitor ModSecurity performance in production. The third part tests the publicly available rule sets in order to give you a taste of what they are like, as well as document a methodology you can use to test your own rules. The chapter then moves to rule set benchmarking, which is an essential part of the process of rule writing. The last part of this chapter gives very practical advice on how to use regular expressions and parallel matching, comparing several approaches and explaining when to use them.

Chapter 11, *Content Injection*, explains how to reach from ModSecurity, which is a server-side tool, right into a user's browser and continue with the inspection there. This feature makes it possible to detect the attacks that were previously thought to be undetectable by a server-side tool, for example DOM-based cross-site scripting attacks. Content injection also comes in handy if you need to communicate with your users—for example, to tell them that they have been attacked.

Chapter 12, *Writing Rules in Lua*, discusses a gem of a feature: writing rules using the Lua programming language. The rule language of ModSecurity is easy to use and can get a lot done, but for the really difficult problems you may need the power of a proper programming language. In addition, you can use Lua to react to events, and it is especially useful when integrating with external systems.

Chapter 13, *Handling XML*, covers the XML capabilities of ModSecurity in detail. You get to learn how to validate XML using either DTDs or XML Schemas, and how to combine XPath expressions with the other features ModSecurity offers to perform both whitelist- and blacklist-based validation. The XML features of ModSecurity have traditionally been poorly documented; here you will find details never covered before. The chapter ends with an XML validation framework you can easily adapt for your needs.

Chapter 14, *Extending Rule Language*, discusses how you can extend ModSecurity to implement new functionality. It gives several step-by-step examples, explaining how to implement a transformation function, an operator, and a variable. Of course, with ModSecurity being

open source, you can extend it directly at any point, but when you use the official APIs, you avoid making a custom version of ModSecurity (which is generally time consuming because it prevents upgrades).

Updates

If you purchased this book directly from Feisty Duck, your purchase includes access to newer digital versions of the book. Updates are made automatically after I update the manuscript, which I keep in DocBook format in a Subversion repository. At the moment, there is a script that runs every hour, and rebuilds the book when necessary. Whenever you visit your personal digital download link, you get the most recent version of the book.

I use a dedicated Twitter account (@modsecuritybook) to announce relevant changes I make to the book. By following that account you'll find out about the improvements pretty much as they happen. You can also follow my personal Twitter account (@ivanristic) or subscribe to my blog, if you are about computer security in general.

In the first two years of its life, I kept *ModSecurity Handbook* up-to-date with every ModSecurity release. There was a full revision in February 2012, which made the book essentially as good and as current as it was on day of the first release back in 2010. Don't take my past performance as a guarantee of what is going to happen in the future, however. At the launch in 2010 I offered a guarantee that the book will be kept up-to-date for at least a year from your purchase. I dropped that promise at the end of 2011, because I could see the possibility that I would stop with the updates at some point. I will keep my promise until the end of 2012, but I don't know what will happen after that.

Feedback

To get in touch with me please write to *ivanr@webkreator.com*. I would like to hear from you very much, because I believe that a book can fulfill its potential only through the interaction among its author(s) and the readers. Your feedback is particularly important when a book is continuously updated, like this one is. When I change the book as a result of your feedback, all the changes are immediately delivered back to you. There is no more waiting for years to see the improvements!

About the Author

Ivan Ristić is a respected security expert and author, known especially for his contribution to the web application firewall field and the development of ModSecurity, the open source web application firewall. He is also the author of *Apache Security*, a comprehensive security guide for the Apache web server. A frequent speaker at computer security conferences, Ivan

is an active participant in the application security community, a member of the Open Web Application Security Project (OWASP), and an officer of the Web Application Security Consortium (WASC).

About the Technical Reviewer

Brian Rectanus is a developer turned manager in the web application security field. He has worked in the past on various security software related projects such as the IronBee open source WAF framework, the ModSecurity open source WAF and the Suricata open source IDS/IPS. Brian is an open source advocate and proud `NIX loving, Mac using, non-Windows user who has been writing code on various `NIX platforms with vi since 1993. Today he still does all his development work in the more modern vim editor—like there is any other—and loves every bit of it. Brian has spent the majority of his career working with web technology from various perspectives, be it manager, developer, administrator or security assessor. Brian has held many certifications in the past, including GCIA and GCIH certification from the SANS Institute and a BS in computer science from Kansas State University.

Acknowledgments

To begin with, I would like to thank the entire ModSecurity community for their support, and especially all of you who used ModSecurity and sent me your feedback. ModSecurity wouldn't be what it is without you. Developing and supporting ModSecurity was a remarkable experience; I hope you enjoy using it as much as I enjoyed developing it.

I would also like to thank my former colleagues from Breach Security, who gave me a warm welcome, even though I joined them pretty late in the game. I regret that, due to my geographic location, I didn't spend more time working with you. I would especially like to thank —in no particular order—Brian Rectanus, Ryan Barnett, Ofer Shezaf, and Avi Aminov, who worked with me on the ModSecurity team. Brian was also kind to work with me on the book as a technical reviewer, and I owe special thanks to him for ensuring I didn't make too many mistakes.

I mustn't forget my copyeditor, Nancy Kotary, who was a pleasure to work with, despite having to deal with DocBook and Subversion, none of which is in the standard copyediting repertoire.

For some reason unknown to me, my dear wife Jelena continues to tolerate my long working hours. Probably because I keep promising to work less, even though that never seems to happen. To her I can only offer my undying love and gratitude for accepting me for who I am. My daughter Iva, who's four, is too young to understand what she means to me, but that's all right —I have the patience to wait for another 20 years or so. She is the other sunshine in my life.

I User Guide

This part, with its 14 chapters, constitutes the main body of the book. The first chapter is the introduction to ModSecurity and your map to the rest of the book. The remaining chapters fall into roughly four groups: installation and configuration, rule writing, practical work, and advanced topics.

1 Introduction

ModSecurity is a tool that will help you secure your web applications. No, scratch that. Actually, ModSecurity is a tool that will help you sleep better at night, and I will explain how. I usually call ModSecurity a *web application firewall* (WAF), because that's the generally accepted term to refer to the class of products that are specifically designed to secure web applications. Other times I will call it an *HTTP intrusion detection tool*, because I think that name better describes what ModSecurity does. Neither name is entirely adequate, yet we don't have a better one. Besides, it doesn't really matter what we call it. The point is that web applications—yours, mine, everyone's—are terribly insecure on average. We struggle to keep up with the security issues and need any help we can get to secure them.

The idea to write ModSecurity came to me during one of my sleepless nights—I couldn't sleep because I was responsible for the security of several web-based products. I could see how most web applications were slapped together with little time spent on design and little time spent on understanding the security issues. Furthermore, not only were web applications insecure, but we had no idea how insecure they were or if they were being attacked. Our only eyes were the web server access and error logs, and they didn't say much.

ModSecurity will help you sleep better at night because, above all, it solves the visibility problem: it lets you see your web traffic. That visibility is key to security: once you are able to see HTTP traffic, you are able to analyze it in real time, record it as necessary, and react to the events. The best part of this concept is that you get to do all of that without actually touching web applications. Even better, the concept can be applied to any application—even if you can't access the source code.

Brief History of ModSecurity

Like many other open source projects, ModSecurity started out as a hobby. Software development had been my primary concern back in 2002, when I realized that producing secure web applications is virtually impossible. As a result, I started to fantasize about a tool that would sit in front of web applications and control the flow of data in and out. The first version was

released in November 2002, but a few more months were needed before the tool became useful. Other people started to learn about it, and the popularity of ModSecurity started to rise.

Initially, most of my effort was spent wrestling with Apache to make request body inspection possible. Apache 1.3.x did not have any interception or filtering APIs, but I was able to trick it into submission. Apache 2.x improved things by providing APIs that do allow content interception, but there was no documentation to speak of. Nick Kew released the excellent *The Apache Modules Book* (Prentice Hall) in 2007, which unfortunately was too late to help me with the development of ModSecurity.

By 2004, I was a changed man. Once primarily a software developer, I became obsessed with web application security and wanted to spend more time working on it. I quit my job and started treating ModSecurity as a business. My big reward came in the summer of 2006, when ModSecurity went head to head with other web application firewalls, in an evaluation conducted by Forrester Research, and came out very favorably. Later that year, my company was acquired by Breach Security. A team of one eventually became a team of many: Brian Rectanus came to work on ModSecurity, Ofer Shezaf took on the rules, and Ryan C. Barnett the community management and education. ModSecurity 2.0, a complete rewrite, was released in late 2006. At the same time we released ModSecurity Community Console, which combined the functionality of a remote logging sensor and a monitoring and reporting GUI.

I stopped being in charge of ModSecurity in January 2009, when I left Breach Security. Brian Rectanus subsequently took the lead. In the meantime, Ryan C. Barnett took charge of the ModSecurity rules and produced a significant improvement with CRS v2. In 2010, Trustwave acquired Breach Security and promised to revitalize ModSecurity. The project is currently run by Ryan C. Barnett and Breno Silva, and there are indeed some signs that the project is getting healthier. I remain involved primarily through my work on this book.

Something spectacular happened in March 2011: Trustwave announced that they would be changing the license of ModSecurity from GPLv2 to Apache Software License (ASLv2). This is a great step toward a wider use of ModSecurity because ASL falls into the category of permissive licenses. Later, the same change was announced for the Core Rule Set project (which is hosted with OWASP).

What Can ModSecurity Do?

ModSecurity is a toolkit for real-time web application monitoring, logging, and access control. I like to think about it as an enabler: there are no hard rules telling you what to do; instead, it is up to you to choose your own path through the available features. That's why the title of this section asks what ModSecurity can do, not what it does.

The freedom to choose what to do is an essential part of ModSecurity's identity and goes very well with its open source nature. With full access to the source code, your freedom to choose

extends to the ability to customize and extend the tool itself to make it fit your needs. It's not a matter of ideology, but of practicality. I simply don't want my tools to restrict what I can do.

Back on the topic of what ModSecurity can do, the following is a list of the most important usage scenarios:

Real-time application security monitoring and access control
At its core, ModSecurity gives you access to the HTTP traffic stream, in real-time, along with the ability to inspect it. This is enough for real-time security monitoring. There's an added dimension of what's possible through ModSecurity's persistent storage mechanism, which enables you to track system elements over time and perform event correlation. You are able to reliably block, if you so wish, because ModSecurity uses full request and response buffering.

Virtual patching
Virtual patching is a concept of vulnerability mitigation in a separate layer, where you get to fix problems in applications without having to touch the applications themselves. Virtual patching is applicable to applications that use any communication protocol, but it is particularly useful with HTTP, because the traffic can generally be well understood by an intermediary device. ModSecurity excels at virtual patching because of its reliable blocking capabilities and the flexible rule language that can be adapted to any need. It is, by far, the activity that requires the least investment, is the easiest activity to perform, and the one that most organizations can benefit from straight away.

Full HTTP traffic logging
Web servers traditionally do very little when it comes to logging for security purposes. They log very little by default, and even with a lot of tweaking you are not able to get everything that you need. I have yet to encounter a web server that is able to log full transaction data. ModSecurity gives you that ability to log anything you need, including raw transaction data, which is essential for forensics. In addition, you get to choose which transactions are logged, which parts of a transaction are logged, and which parts are sanitized.

Continuous passive security assessment
Security assessment is largely seen as an active scheduled event, in which an independent team is sourced to try to perform a simulated attack. Continuous passive security assessment is a variation of real-time monitoring, where, instead of focusing on the behavior of the external parties, you focus on the behavior of the system itself. It's an early warning system of sorts that can detect traces of many abnormalities and security weaknesses before they are exploited.

Web application hardening
One of my favorite uses for ModSecurity is attack surface reduction, in which you selectively narrow down the HTTP features you are willing to accept (e.g., request methods, request headers, content types, etc.). ModSecurity can assist you in enforcing many

similar restrictions, either directly, or through collaboration with other Apache modules. They all fall under web application hardening. For example, it is possible to fix many session management issues, as well as cross-site request forgery vulnerabilities.

Something small, yet very important to you

Real life often throws unusual demands to us, and that is when the flexibility of ModSecurity comes in handy where you need it the most. It may be a security need, but it may also be something completely different. For example, some people use ModSecurity as an XML web service router, combining its ability to parse XML and apply XPath expressions with its ability to proxy requests. Who knew?

> **Note**
>
> I often get asked if ModSecurity can be used to protect Apache itself. The answer is that it can, in some limited circumstances, but that it isn't what it is designed for. You may sometimes be able to catch an attack with ModSecurity before it hits a vulnerable spot in Apache or in a third-party module, but there's a large quantity of code that runs before ModSecurity. If there's a vulnerability in that area, ModSecurity won't be able to do anything about it.

What Are Web Application Firewalls, Anyway?

I said that ModSecurity is a web application firewall, but it's a little known fact that no one really knows what web application firewalls are. It is generally understood that a web application firewall is an intermediary element (implemented either as a software add-on or process, or as a network device) that enhances the security of web applications, but opinions differ once you dig deeper. There are many theories that try to explain the different views, but the best one I could come up with is that, unlike anything we had before, the web application space is so complex that there is no easy way to classify what we do security-wise. Rather than focus on the name, you should focus on what a particular tool does and how it can help.

If you want to learn more about the topic, there are two efforts that focus on understanding web application firewalls:

- *Web application firewall evaluation criteria* (WAFEC) is a project of the Web Application Security Consortium (WASC). It's an older effort (which has been inactive for a couple of years now) that focuses on the technical features of web application firewalls.

- *Best practices: Web Application Firewalls* is a project of Open Web Application Security Project (OWASP) that focuses largely on the practicalities of WAF deployment, which is an important aspect that is often overlooked.

Guiding Principles

There are four guiding principles on which ModSecurity is based, as follows:

Flexibility

I think that it's fair to say that I built ModSecurity for myself: a security expert who needs to intercept, analyze, and store HTTP traffic. I didn't see much value in hard-coded functionality, because real life is so complex that everyone needs to do things just slightly differently. ModSecurity achieves flexibility by giving you a powerful rule language, which allows you to do exactly what you need to, in combination with the ability to apply rules only where you need to.

Passiveness

ModSecurity will take great care to never interact with a transaction unless you tell it to. That is simply because I don't trust tools, even the one I built, to make decisions for me. That's why ModSecurity will give you plenty of information, but ultimately leave the decisions to you.

Predictability

There's no such thing as a perfect tool, but a predictable one is the next best thing. Armed with all the facts, you can understand ModSecurity's weak points and work around them.

Quality over quantity

Over the course of six years spent working on ModSecurity, we came up with many ideas for what ModSecurity could do. We didn't act on most of them. We kept them for later. Why? Because we understood that we have limited resources available at our disposal and that our minds (ideas) are far faster than our implementation abilities. We chose to limit the available functionality, but do really well at what we decided to keep in.

There are bits in ModSecurity that fall outside the scope of these four principles. For example, ModSecurity can change the way Apache identifies itself to the outside world, confine the Apache process within a jail, and even implement an elaborate scheme to deal with a once-infamous universal XSS vulnerability in Adobe Reader. Although it was I who added those features, I now think that they detract from the main purpose of ModSecurity, which is a reliable and predictable tool that allows for HTTP traffic inspection.

Deployment Options

ModSecurity supports two deployment options: embedded and reverse proxy deployment. There is no one correct way to use them; choose an option based on what best suits your circumstances. There are advantages and disadvantages to both options:

Embedded

Because ModSecurity is an Apache module, you can add it to any compatible version of Apache. At the moment that means a reasonably recent Apache version from the 2.0.x branch, although a newer 2.2.x version is recommended. The embedded option

is a great choice for those who already have their architecture laid out and don't want to change it. Embedded deployment is also the only option if you need to protect hundreds of web servers. In such situations, it is impractical to build a separate proxy-based security layer. Embedded ModSecurity not only does not introduce new points of failure, but it scales seamlessly as the underlying web infrastructure scales. The main challenge with embedded deployment is that server resources are shared between the web server and ModSecurity.

Reverse proxy

Reverse proxies are effectively HTTP routers, designed to stand between web servers and their clients. When you install a dedicated Apache reverse proxy and add ModSecurity to it, you get a "proper" network web application firewall, which you can use to protect any number of web servers on the same network. Many security practitioners prefer having a separate security layer. With it you get complete isolation from the systems you are protecting. On the performance front, a standalone ModSecurity will have resources dedicated to it, which means that you will be able to do more (i.e., have more complex rules). The main disadvantage of this approach is the new point of failure, which will need to be addressed with a high-availability setup of two or more reverse proxies.

Is Anything Missing?

ModSecurity is a very good tool, but there are a number of features, big and small, that could be added. The small features are those that would make your life with ModSecurity easier, perhaps automating some of the boring work (e.g., persistent blocking, which you now have to do manually). But there are really only two features that I would call missing:

Learning

Defending web applications is difficult, because there are so many of them, and they are all different. (I often say that every web application effectively creates its own communication protocol.) It would be very handy to have ModSecurity observe application traffic and create a model that could later be used to generate policy or assist with false positives. While I was at Breach Security, I started a project called ModProfiler as a step toward learning, but that project is still as I left it, as version 0.2.

Passive mode of deployment

ModSecurity can be embedded only in Apache 2.x, but when you deploy it as a reverse proxy, it can be used to protect any web server. Reverse proxies are not everyone's cup of tea, however, and sometimes it would be very handy to deploy ModSecurity passively, without having to change anything on the network.

Although a GUI is not within the scope of the project, there are currently two options when it comes to remote logging and alert management. You will find them in the Resources section later in this chapter.

Getting Started

In this first practical section in the book, I will give you a whirlwind tour of the ModSecurity internals, which should help you get started.

Hybrid Nature of ModSecurity

ModSecurity is a hybrid web application firewall engine that relies on the host web server for some of the work. The only supported web server at the moment is Apache 2.x, but it is possible, in principle, to integrate ModSecurity with any other web server that provides sufficient integration APIs.

Apache does for ModSecurity what it does for all other modules—it handles the infrastructure tasks:

1. Decrypts SSL
2. Breaks up the inbound connection stream into HTTP requests
3. Partially parses HTTP requests
4. Invokes ModSecurity, choosing the correct configuration context (`<VirtualHost>`, `<Location>`, etc.)
5. De-chunks request bodies as necessary

There a few additional tasks Apache performs in a reverse proxy scenario:

1. Forwards requests to backend servers (with or without SSL)
2. Partially parses HTTP responses
3. De-chunks response bodies as necessary

The advantage of a hybrid implementation is that it is very efficient—the duplication of work is minimal when it comes to HTTP parsing. A couple of disadvantages of this approach are that you don't always get access to the raw data stream and that web servers sometimes don't process data in the way a security-conscious tool would. In the case of Apache, the hybrid approach works reasonably well, with a few minor issues:

Request line and headers are NUL-terminated
This is normally not a problem, because what Apache doesn't see cannot harm any module or application. In some very rare cases, however, the purpose of the NUL-byte evasion is to hide things, and this Apache behavior only helps with the hiding.

Request header transformation
Apache will canonicalize request headers, combining multiple headers that use the same name and collapsing those that span two or more lines. The transformation may make it difficult to detect subtle signs of evasion, but in practice this hasn't been a problem yet.

Quick request handling

Apache will handle some requests quickly, leaving ModSecurity unable to do anything but notice them in the logging phase. Invalid HTTP requests, in particular, will be rejected by Apache without ModSecurity having a say.

No access to some response headers

Because of the way Apache works, the `Server` and `Date` response headers are invisible to ModSecurity; they cannot be inspected or logged.

Main Areas of Functionality

The functionality offered by ModSecurity falls roughly into four areas:

Parsing

ModSecurity tries to make sense of as much data as available. The supported data formats are backed by security-conscious parsers that extract bits of data and store them for use in the rules.

Buffering

In a typical installation, both request and response bodies will be buffered. This means that ModSecurity usually sees complete requests before they are passed to the application for processing, and complete responses before they are sent to clients. Buffering is an important feature, because it is the only way to provide reliable blocking. The downside of buffering is that it requires additional RAM to store the request and response body data.

Logging

Full transaction logging (also referred to as *audit logging*) is a big part of what ModSecurity does. This feature allows you to record complete HTTP traffic, instead of just rudimentary access log information. Request headers, request body, response header, response body—all those bits will be available to you. It is only with the ability to see what is happening that you will be able to stay in control.

Rule engine

The rule engine builds on the work performed by all other components. By the time the rule engine starts operating, the various bits and pieces of data it requires will all be prepared and ready for inspection. At that point, the rules will take over to assess the transaction and take actions as necessary.

> **Note**
>
> There's one thing ModSecurity purposefully avoids to do: as a matter of design, ModSecurity does not support data sanitization. I don't believe in sanitization, purely because I believe that it is too difficult to get right. If you know for sure that you are being attacked (as you have to before you can decide to sanitize), then you should refuse

to process the offending requests altogether. Attempting to sanitize merely opens a new battlefield where your attackers don't have anything to lose, but everything to win. You, on the other hand, don't have anything to win, but everything to lose.

What Rules Look Like

Everything in ModSecurity revolves around two things: configuration and rules. The configuration tells ModSecurity how to process the data it sees; the rules decide what to do with the processed data. Although it is too early to go into how the rules work, I will show you a quick example here just to give you an idea what they look like.

For example:

```
SecRule ARGS "<script>" log,deny,status:404
```

Even without further assistance, you can probably recognize the part in the rule that specifies what we wish to look for in input data (`<script>`). Similarly, you will easily figure out what will happen if we do find the desired pattern (`log,deny,status:404`). Things will become more clear if I tell you about the general rule syntax, which is the following:

```
SecRule VARIABLES OPERATOR ACTIONS
```

The three parts have the following meanings:

1. The `VARIABLES` part tells ModSecurity where to look. The `ARGS` variable, used in the example, means all request parameters.

2. The `OPERATOR` part tells ModSecurity how to look. In the example, we have a regular expression pattern, which will be matched against `ARGS`.

3. The `ACTIONS` part tells ModSecurity what to do on a match. The rule in the example gives three instructions: `log` problem, `deny` transaction and use the status 404 for the denial (`status:404`).

I hope you are not disappointed with the simplicity of this first rule. I promise you that by combining the various facilities offered by ModSecurity, you will be able to write very useful rules that implement complex logic where necessary.

Transaction Lifecycle

In ModSecurity, every transaction goes through five steps, or phases. In each of the phases, ModSecurity will do some work at the beginning (e.g., parse data that has become available), invoke the rules specified to work in that phase, and perhaps do a thing or two after the phase rules have finished. At first glance, it may seem that five phases are too many, but there's a reason why each of the phases exist. There is always one thing, sometimes several, that can only be done at a particular moment in the transaction lifecycle.

Request headers (1)

The request headers phase is the first entry point for ModSecurity. The principal purpose of this phase is to allow rule writers to assess a request before the costly request body processing is undertaken. Similarly, there is often a need to influence how ModSecurity will process a request body, and this phase is the place to do it. For example, ModSecurity will not parse an XML request body by default, but you can instruct it do so by placing the appropriate rules into phase 1.

Request body (2)

The request body phase is the main request analysis phase and takes place immediately after a complete request body has been received and processed. The rules in this phase have all the available request data at their disposal.

Response headers (3)

The response headers phase takes place after response headers become available, but before a response body is read. The rules that need to decide whether to inspect a response body should run in this phase.

Response body (4)

The response body phase is the main response analysis phase. By the time this phase begins, the response body will have been read, with all its data available for the rules to make their decisions.

Logging (5)

The logging phase is special in more ways than one. First, it's the only phase from which you cannot block. By the time this phase runs, the transaction will have finished, so there's little you can do but record the fact that it happened. Rules in this phase are run to control how logging is done.

Lifecycle Example

To give you a better idea what happens on every transaction, we'll examine a detailed debug log of one POST transaction. I've deliberately chosen a transaction type that uses the request body as its principal method to transmit data, because following such a transaction will exercise most parts of ModSecurity. To keep things relatively simple, I used a configuration without any rules, removed some of the debug log lines for clarity, and removed the timestamps and some additional metadata from each line.

> **Note**
>
> Please do not try to understand everything about the logs at this point. The idea is just to get a general feel about how ModSecurity works, and to introduce you to debug logs. Very quickly after starting to use ModSecurity, you will discover that the debug logs will be an indispensable rule writing and troubleshooting tool.

The transaction I am using as an example in this section is very straightforward. I made a point of placing request data in two different places, parameter *a* in the query string and parameter *b* in the request body, but there is little else of interest in the request:

```
POST /?a=test HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 6

b=test
```

The response is entirely unremarkable:

```
HTTP/1.1 200 OK
Date: Sun, 17 Jan 2010 00:13:44 GMT
Server: Apache
Content-Length: 12
Connection: close
Content-Type: text/html

Hello World!
```

ModSecurity is first invoked by Apache after request headers become available, but before a request body (if any) is read. First comes the initialization message, which contains the unique transaction ID generated by mod_unique_id. Using this information, you should be able to pair the information in the debug log with the information in your access and audit logs. At this point, ModSecurity will parse the information on the request line and in the request headers. In this example, the query string part contains a single parameter (a), so you will see a message documenting its discovery. ModSecurity will then create a transaction context and invoke the REQUEST_HEADERS phase:

```
[4] Initialising transaction (txid SopXW38EAAE9YbLQ).
[5] Adding request argument (QUERY_STRING): name "a", value "test"
[4] Transaction context created (dcfg 8121800).
[4] Starting phase REQUEST_HEADERS.
```

Assuming that a rule didn't block the transaction, ModSecurity will now return control to Apache, allowing other modules to process the request before control is given back to it.

In the second phase, ModSecurity will first read and process the request body, if it is present. In the following example, you can see three messages from the input filter, which tell you what was read. The fourth message tells you that one parameter was extracted from the request body. The content type used in this request (application/x-www-form-urlencoded) is one of the types ModSecurity recognizes and parses automatically. Once the request body is processed, the REQUEST_BODY rules are processed.

```
[4] Second phase starting (dcfg 8121800).
```

```
[4] Input filter: Reading request body.
[9] Input filter: Bucket type HEAP contains 6 bytes.
[9] Input filter: Bucket type EOS contains 0 bytes.
[5] Adding request argument (BODY): name "b", value "test"
[4] Input filter: Completed receiving request body (length 6).
[4] Starting phase REQUEST_BODY.
```

The filters that keep being mentioned in the logs are parts of ModSecurity that handle request and response bodies:

```
[4] Hook insert_filter: Adding input forwarding filter (r 81d0588).
[4] Hook insert_filter: Adding output filter (r 81d0588).
```

There will be a message in the debug log every time ModSecurity sends a chunk of data to the request handler, and one final message to say that there isn't any more data in the buffers.

```
[4] Input filter: Forwarding input: mode=0, block=0, nbytes=8192 ↵
(f 81d2228, r 81d0588).
[4] Input filter: Forwarded 6 bytes.
[4] Input filter: Sent EOS.
[4] Input filter: Input forwarding complete.
```

Shortly thereafter, the output filter will start receiving data, at which point the RESPONSE_HEADERS rules will be invoked:

```
[9] Output filter: Receiving output (f 81d2258, r 81d0588).
[4] Starting phase RESPONSE_HEADERS.
```

Once all the rules have run, ModSecurity will continue to store the response body in its buffers, after which it will run the RESPONSE_BODY rules:

```
[9] Output filter: Bucket type MMAP contains 12 bytes.
[9] Output filter: Bucket type EOS contains 0 bytes.
[4] Output filter: Completed receiving response body (buffered full - 12 bytes).
[4] Starting phase RESPONSE_BODY.
```

Again, assuming that none of the rules blocked, the accumulated response body will be forwarded to the client:

```
[4] Output filter: Output forwarding complete.
```

Finally, the logging phase will commence. The LOGGING rules will be run first to allow them to influence logging, after which the audit logging subsystem will be invoked to log the transaction if necessary. A message from the audit logging subsystem will be the last transaction message in the logs. In this example, ModSecurity tells us that it didn't find anything of interest in the transaction and that it sees no reason to log it:

```
[4] Initialising logging.
```

```
[4] Starting phase LOGGING.
[4] Audit log: Ignoring a non-relevant request.
```

File Upload Example

Requests that contain files are processed slightly differently. The changes can be best understood by again following the activity in the debug log:

```
[4] Input filter: Reading request body.
[9] Multipart: Boundary: --------------------------2411583925858
[9] Input filter: Bucket type HEAP contains 256 bytes.
[9] Multipart: Added part header "Content-Disposition" "form-data; name=\"f\"; ↵
filename=\"eicar.com.txt\""
[9] Multipart: Added part header "Content-Type" "text/plain"
[9] Multipart: Content-Disposition name: f
[9] Multipart: Content-Disposition filename: eicar.com.txt
[4] Multipart: Created temporary file: ↵
/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF
[9] Multipart: Changing file mode to 0600: ↵
/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF
[9] Multipart: Added file part 9c870b8 to the list: name "f" file name ↵
"eicar.com.txt" (offset 140, length 68)
[9] Input filter: Bucket type EOS contains 0 bytes.
[4] Reqest body no files length: 96
[4] Input filter: Completed receiving request body (length 256).
```

In addition to seeing the multipart parser in action, you see ModSecurity creating a temporary file (into which it will extract the upload) and adjusting its privileges to match the desired configuration.

Then, at the end of the transaction, you will see the cleanup and the temporary file deleted:

```
[4] Multipart: Cleanup started (remove files 1).
[4] Multipart: Deleted file (part) ↵
"/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF"
```

The temporary file will not be deleted if ModSecurity decides to keep an uploaded file. Instead, it will be moved to the storage area:

```
[4] Multipart: Cleanup started (remove files 0).
[4] Input filter: Moved file from ↵
"/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF" to ↵
"/opt/modsecurity/var/upload/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF".
```

In the example traces, you've observed an upload of a small file that was stored in RAM. When large uploads take place, ModSecurity will attempt to use RAM at first, switching to on-disk storage once it becomes obvious that the file is larger:

```
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 1536 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 576 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[4] Input filter: Request too large to store in memory, switching to disk.
```

A new file will be created to store the entire raw request body:

```
[4] Input filter: Created temporary file to store request body: ↵
/opt/modsecurity/var/tmp//20090819-180105-Sowv0X8AAQEAACWAArs-request_body-4nZjqf
[4] Input filter: Wrote 129559 bytes from memory to disk.
```

This file is always deleted in the cleanup phase:

```
[4] Input filter: Removed temporary file: ↵
/opt/modsecurity/var/tmp//20090819-180105-Sowv0X8AAQEAACWAArs-request_body-4nZjqf
```

Impact on Web Server

The addition of ModSecurity will change how your web server operates. As with all Apache modules, you pay for the additional flexibility and security ModSecurity gives you with increased CPU and RAM consumption on your server. The exact amount will depend on your configuration of ModSecurity and the usage of your server. Following is a detailed list of the various activities that increase resource consumption:

- ModSecurity will add to the parsing already done by Apache, and that results in a slight increase of CPU consumption.

- Complex parsers (e.g., XML) are more expensive.

- The handling of file uploads may require I/O operations. In some cases, inbound data will be duplicated on disk.

- The parsing will add to the RAM consumption, because every extracted element (e.g., a request parameter) will need to be copied into its own space.

- Request bodies and response bodies are usually buffered in order to support reliable blocking.
- Every rule in your configuration will use some of the CPU time (for the operator) and RAM (to transform input data before it can be analyzed).
- Some of the operators used in the rules (e.g., the regular expression operator) are CPU-intensive.
- Full transaction logging is an expensive I/O operation.

In practice, this list is important because it keeps you informed; what matters is that you have enough resources to support your ModSecurity needs. If you do, then it doesn't matter how expensive ModSecurity is. Also, what's expensive to someone may not be to someone else. If you don't have enough resources to do everything you want with ModSecurity, you will need to monitor the operation of your system and remove some of the functionality to reduce the resource consumption. Virtually everything that ModSecurity does is configurable, so you should have no problems doing that.

It is generally easier to run ModSecurity in reverse proxy mode, because then you usually have an entire server (with its own CPU and RAM) to play with. In embedded mode, ModSecurity will add to the processing already done by the web server, so this method is more challenging on a busy server.

For what it's worth, ModSecurity generally uses the minimal necessary resources to perform the desired functions, so this is really a case of exchanging functionality for speed: if you want to do more, you have to pay more.

What Next?

The purpose of this section is to map your future ModSecurity activities and help you determine where to go from here. Where you will go depends on what you want to achieve and how much time you have to spend. A complete ModSecurity experience, so to speak, consists of the following elements:

Installation and configuration
This is the basic step that all users must learn how to perform. The next three chapters will teach you how to make ModSecurity operational, performing installation, general configuration, and logging configuration. Once you are done with that, you need to decide what you want to do with it. That's what the remainder of the book is for.

Rule writing
Rule writing is an essential skill. You may currently view rules as a tool to use to detect application security attacks. They are that, but they are also much more. In ModSecurity, you write rules to find out more about HTTP clients (e.g., geolocation and IP address reputation), perform long-term activity tracking (of IP addresses, sessions

and users, for example), implement policy decisions (use the available information to make the decisions to warn or block), write virtual patches, and even to check on the status of ModSecurity itself.

It is true that the attack detection rules are in a class of its own, but that's mostly because, in order to write them successfully, you need to know so much about application security. For that reason, many ModSecurity users generally focus on using third-party rule sets for the attack detection. It's a legitimate choice. Not everyone has the time and inclination to become an application security expert. Even if you end up not using any inspection rules whatsoever, the ability to write virtual patches is reason enough to use ModSecurity.

Rule sets

The use of existing rule sets is the easiest way to get to the proverbial low hanging fruit: invest small effort and reap big benefits. Traditionally, the main source of ModSecurity rules has been the Core Rule Set project, now hosted with OWASP. On the other hand, if you are keen to get your hands dirty, I can tell you that I draw great pleasure from writing my own rules. It's a great way to learn about application security. The only drawback is that it requires a large time investment.

Remote logging and alert management GUI

ModSecurity is perfectly usable without a remote logging solution and without a GUI (the two usually go together). Significant error messages are copied to Apache's error log. Complete transactions are usually logged to the audit log. With a notification system in place, you will know when something happens, and you can visit the audit logs to investigate. For example, many installations will divert Apache's error log to a central logging system (via syslog).

The process does become more difficult with more than one sensor to manage. Furthermore, GUIs make the whole experience of monitoring much more pleasant. For that reason you will probably seek to install one of the available remote centralization tools and use their GUIs. The available options are listed in the Resources section, which follows.

Resources

This section contains a list of assorted ModSecurity resources that can assist you in your work.

Figure 2-1. The homepage of www.modsecurity.org

General Resources

The following resources are the bare essentials:

ModSecurity web site

ModSecurity's web site is probably going to be your main source of information. You should visit the web site from time to time, as well as subscribe to receive the updates from the blog.

Official documentation

The official ModSecurity documentation is maintained in a wiki, but copies of it are made for inclusion with every release.

Issue tracker

The ModSecurity issue tracker is the place you will want to visit for one of two reasons: to report a problem with ModSecurity itself (e.g., when you find a bug) or to check out the progress on the next (major or minor) version. Before reporting any problems, go through the Support Checklist, which will help you assemble the information required

to help resolve your problem. Providing as much information as you can will help the developers understand and replicate the problem, and provide a fix (or a workaround) quickly.

Users' mailing list

The users' mailing list (mod-security-users@lists.sourceforge.net) is a general-purpose mailing list where you can discuss ModSecurity. Feel free to ask questions, propose improvements, and discuss ideas. That is the place where you'll hear first about new ModSecurity versions.

ModSecurity@Freshmeat

If you subscribe to the users' mailing list, you will generally find out about new versions of ModSecurity as soon as they are released. If you care only about version releases, however, you may consider subscribing to the new version notifications at the ModSecurity page at Freshmeat.

Core Rules mailing list

Starting with version 2, the Core Rules project is part of OWASP, and has a separate mailing list (owasp-modsecurity-core-rule-set@lists.owasp.org).

Developer Resources

If you are interested in development work, you will need these:

Developers' mailing list

The developers' mailing list is generally a lonely place, but if you do decide to start playing with the ModSecurity source code, this list is the place to go to discuss your work.

Source code access

The source code of ModSecurity is hosted at a Subversion repository at SourceForge, which allows you to access it directly or through a web-based user interface.

FishEye interface

If you are not looking to start developing immediately but still want to have a look at the source code of ModSecurity, I recommend that you use the ModSecurity FishEye interface, which is much better than the stock interface available at SourceForge.

AuditConsole

Using ModSecurity entirely from the command line is possible but not much fun. The configuration part is not a problem, but reviewing logs is difficult without higher-level tools. Your best choice for a log centralization and GUI tool is AuditConsole, which is built by Christian Bockermann and hosted on *www.jwall.org*.

AuditConsole is free and provides the following features:

- Event centralization from multiple remote ModSecurity installations
- Event storage and retrieval
- Support for multiple user accounts and support for different views
- Event tagging
- Event rules, which are executed in the console

Summary

This chapter was your ModSecurity orientation. I introduced ModSecurity at a high level, discussed what it is and what it isn't, and what it can do and what it cannot. I also gave you a taste of what ModSecurity is like and described common usage scenarios, as well as covered some of the interesting parts of its operation.

The foundation you now have should be enough to help you set off on a journey of ModSecurity exploration. The next chapter discusses installation.

2 Installation

Before you can install ModSecurity, you need to decide if you want to compile it from source or use a binary version—either one included with your operating system or one produced by a third party. Each of the options comes with its advantages and disadvantages, as listed in Table 2.1, "Installation options".

Table 2.1. Installation options

Installation type	Advantages	Disadvantages
Operating system version	• Fully automated installation • Maintenance included	• May not be the latest version
Third-party binary	• Semi-automated installation	• May not be the latest version • Manual download and updates • Do you trust the third party?
Source code	• Can always use the latest version • Can use experimental versions • Can make changes, apply patches, and make emergency security fixes	• Manual installation and maintenance required • A lot of work involved with rolling your own version

In some cases, you won't have a choice. For example, if you've installed Apache from source, you will need to install ModSecurity from source too (you will be able to reuse the system packages, of course). The following questions may help you to make the decision:

- Do you intend to use ModSecurity seriously?

- Are you comfortable compiling programs from source?

- Do you have enough time to spend on the compilation and the successive maintenance of a custom-installed program?

- Will you need to make changes to ModSecurity, or write your own extensions?

I generally try to use binary packages when they are available (and they are available on De-bian, which is currently my platform of choice). When I build dedicated reverse proxy installations, however, I tend to build everything from source, because that allows me access to the latest Apache and ModSecurity versions, and makes it easier to tweak things (by changing the source code of either Apache or ModSecurity) when I want to.

Installation from Source

Installing from source is the preferred approach to installing ModSecurity, mostly because that way you get the latest (and best) version, and because you are able to make any changes you want.

Downloading Releases

To download ModSecurity, go to its web site or the SourceForge project page. You will need both the main distribution and the cryptographic signature:

```
$ wget http://www.modsecurity.org/download/modsecurity-apache_2.5.10-dev2.tar.gz
$ wget http://www.modsecurity.org/download/modsecurity-apache_2.5.10-dev2.tar.gz.asc
```

Verify the signature before doing anything else. That will ensure that the package you've just downloaded does not contain a trojan horse planted by a third party and that it hasn't been corrupted during transport.

```
$ gpg --verify modsecurity-apache_2.5.10-dev2.tar.gz.asc
gpg: Signature made Wed 12 Aug 2009 23:27:06 BST using DSA key ID E77B534D
gpg: Can't check signature: public key not found
```

Your first attempt may not provide the expected results, but that can be solved easily by importing the referenced key from a key server:

```
$ gpg --recv-keys E77B534D
gpg: requesting key E77B534D from hkp server keys.gnupg.net
gpg: /home/guest/.gnupg/trustdb.gpg: trustdb created
gpg: key E77B534D: public key "Brian Rectanus (work) <brian.rectanus@breach.com>" ↵
imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:               imported: 1
```

Now you can try again:

```
$ gpg --verify modsecurity-apache_2.5.10-dev2.tar.gz.asc
gpg: Signature made Wed 12 Aug 2009 23:27:06 BST using DSA key ID E77B534D
```

```
gpg: Good signature from "Brian Rectanus (work) <brian.rectanus@breach.com>"
gpg:                 aka "Brian Rectanus <brian@rectanus.net>"
gpg:                 aka "Brian Rectanus (personal) <brectanu@gmail.com>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
```

While this warning might look serious, it generally isn't a problem, and has to do with the way gpg expects you to verify the identity of an individual. The warning basically tells you that you've downloaded Brian's key from somewhere, but that you don't *really know* that it belongs to him. The only way to be sure, as far as gpg is concerned, is to meet Brian in real life, or to meet someone else who knows him personally. If you want to learn more, look up *web of trust* on Wikipedia.

Downloading from Repository

If you want to be on the cutting edge, downloading the latest development version directly from the Subversion (the source code control system used by the ModSecurity project) repository is the way to go. When you do this, you'll get new features days and even months before they make it into an official stable release. Having said that, however, there is a reason why we call some versions "stable." When you use a repository version of ModSecurity, you need to accept that there is no guarantee whatsoever that it will work correctly. For what it's worth, I am currently running a development version in production, and I am confident that it will not bring my server down.

Before you can install a development version of ModSecurity, you need to know where to find it. The repository, which is hosted with SourceForge, can be viewed with a browser. The view of the root of the repository is similar to that in Figure 2-1, "ModSecurity repository root".

Figure 2-1. ModSecurity repository root
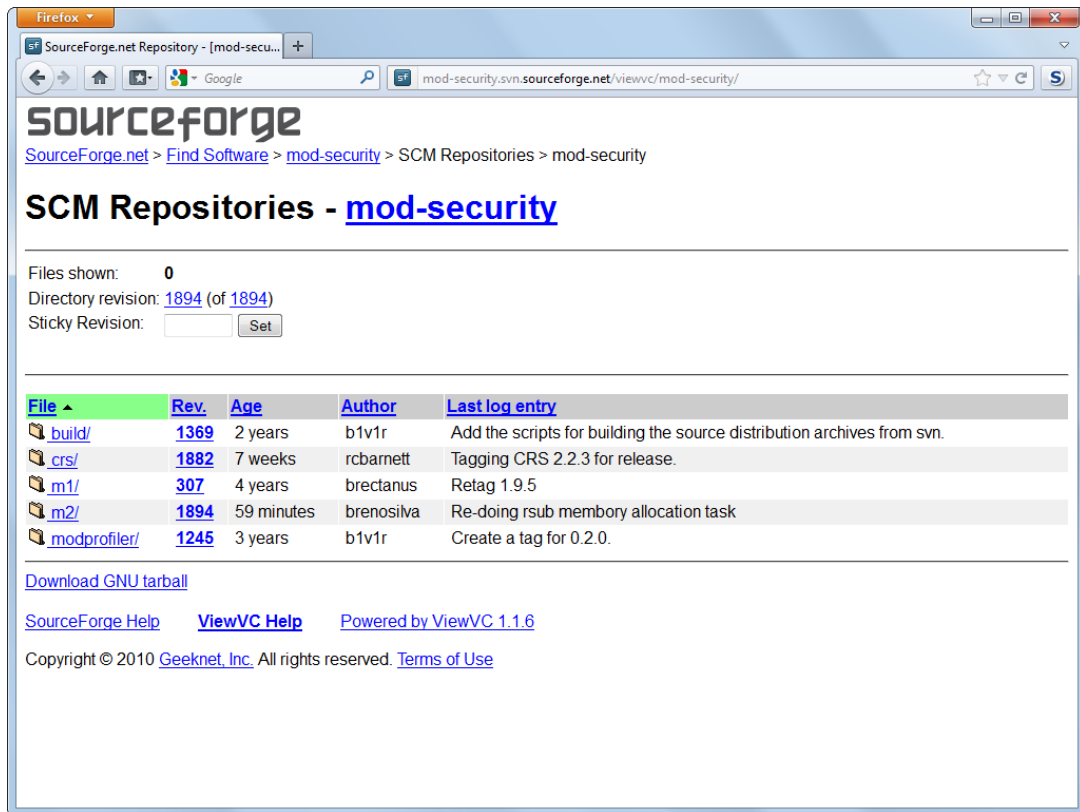
What you probably want is located in the `m2/` directory, which houses ModSecurity 2.x. Within that directory, you will find a directory structure that you will find familiar if you've worked with Subversion before:

```
m2/
    branches/
        2.1.x/
        2.5.x/
        2.6.x/
    experimental/
        [some stuff you won't care about]
    tags/
        [all releases, one per directory]
    trunk/
```

The `trunk` directory always contains the most recent development version. The active branches may sometimes contain a feature or a fix that has not been released yet. The branches will always be generally stable anyway, and the risk of something breaking is minimal.

Once you have determined the location of the version of ModSecurity that you wish to use, you can get it using the export command of Subversion, like this:

```
$ svn export https://mod-security.svn.sourceforge.net/svnroot/mod-security/↵
m2/trunk modsecurity-trunk
```

What you will get in the folder modsecurity-trunk is almost the same as what you get when you download a release. The documentation might not be in sync, however. Starting with ModSecurity 2.6, the master documentation is kept in a wiki, with copies of the wiki included with releases.

Installation on Unix

Before you can start to compile ModSecurity, you must ensure that you have a complete development toolchain installed. Refer to the documentation of the operating system you are using for instructions. If you'll be adding ModSecurity to an operating system–provided Apache, you are likely to need to install a specific Apache development package, too. For example, on Debian and Ubuntu, you need to use apache2-prefork-dev or apache2-threaded-dev, depending on which deployment model (process-based or thread-based) you chose.

In the next step, ensure that you have resolved all the dependencies before compilation. The dependencies are listed in Table 2.2, "ModSecurity dependencies".

Table 2.2. ModSecurity dependencies

Dependency	In Apache?	Purpose
Apache Portable Runtime	Yes	Various
APR-Util	Yes	Various
mod_unique_id	Yes, but may not be installed by default	Generate unique transaction ID
libcurl	No	Remote logging (mlogc)
libxml2	No	XML processing
Lua 5.1	No	Writing complex rules in Lua (optional)
PCRE	Yes, but cannot be used by ModSecurity	Regular expression matching

If you already have Apache installed, you will only ever need to deal with libcurl, libxml2, and Lua. With Apache compiled from source, you will also need the PCRE library. Although Apache comes bundled with one, it is used in a way that does not allow other modules to access it. To work around this issue, install PCRE separately and then tell Apache to use the external copy. I explain how to do that later in this section.

If you're installing from source, go to the packages' web sites, and download and install the tarballs. If you're using managed packages, you just need to determine what the missing packages are called. On Debian Lenny, the following command installs the missing packages:

```
# apt-get install libcurl3-dev liblua5.1-dev libxml2-dev
```

Refer to the documentation of the package management system used by your platform to determine how to search the package database.

> **Note**
>
> Libcurl, which is used for remote logging, can be compiled to use OpenSSL or GnuT-LS. You are advised to use OpenSSL because there have been complaints about remote logging problems when GnuTLS was used.

The process should be straightforward from here on. Execute the following commands in succession:

```
$ ./configure
$ make
```

This set of commands assumes that you don't need any compile-time options. If you do, see the following section. Before ModSecurity 2.6, the making of `mlogc` was optional and you had to execute another `make mlogc` to do it.

> **Note**
>
> Running additional tests after compilation (`make test` and `make test-regression`) is always a good idea, and is an especially good idea when using a development version of ModSecurity. If you are going to have any problems, you want to have them before installation, rather than after.

After ModSecurity is built, one more step is required to install it:

```
$ sudo make install
```

This command adds the module to your Apache installation, but does not activate it—you must do that manually. (While you are there, confirm that `mod_unique_id` is enabled; ModSecurity requires it.) The command will also create a folder (`/usr/local/modsecurity` by default) and store the various run-time files in it. Here's what you get:

```
bin/
    mlogc
    mlogc-batch-load.pl
    rules-updater.pl
lib/
    mod_security2.so
```

Compile-Time Options

The configuration example from the previous section assumed that the dependencies were all installed as system libraries. It also assumed that the configure script will figure everything out on its own. It may or may not, but chances are good that you will occasionally need to do something different; this is where the compile-time options, listed in Table 2.3, "Main compile-time options", come in handy.

Table 2.3. Main compile-time options

Option	Description
--with-apr	Specify location of the Apache Portable Runtime library.
--with-apu	Specify location of the APR-Util library.
--with-apxs	Specify the location of Apache through the location of the apxs script.
--with-curl	Specify the location of libcurl.
--with-libxml	Specify the location of libxml2. Some older versions used --with-xml instead.
--with-pcre	Specify the location of PCRE.

Custom-Compiled Apache Installations

Using ModSecurity with a custom-compiled version of Apache is more work than it should be because of a dependency problem that we have to resolve manually. PCRE, which ModSecurity uses for pattern matching, is a very popular library that is integrated into many operating systems. The problem is that Apache bundles PCRE and uses the bundled version by default when you compile it from source. Unless you take care to avoid conflicts, you'll probably end up with Apache using the bundled version of PCRE and ModSecurity using the one provided by the operating system. The solution to this problem is to build Apache to use the external PCRE version too, which requires just one change to the way you build Apache.

The configure option you need (remember, this is the configure of Apache, not ModSecurity) is --with-pcre, and it is used like this:

```
$ ./configure \
    --with-pcre=/usr/bin/pcre-config \
    --enable-unique-id
```

We are also enabling the mod_unique_id module, which is required for ModSecurity's full transaction log. Now you need to again compile and install Apache.

To configure ModSecurity, use the --with-apxs compile-time option to specify the location of your Apache installation. In the following example, I am assuming Apache is installed in /home/ivanr/apache:

```
$ ./configure \
    --with-apxs=/home/ivanr/apache/bin/apxs
```

From here you install ModSecurity as described in the previous section.

After both Apache and ModSecurity are installed, you should confirm that both products link to the same PCRE library, using `ldd`:

```
$ ldd ~/apache/bin/httpd  | grep pcre
        libpcre.so.3 => /usr/lib/libpcre.so.3 (0xb7d3f000)
```

You should get the same result when you compile ModSecurity:

```
$ ldd ~/apache/modules/mod_security2.so | grep pcre
        libpcre.so.3 => /usr/lib/libpcre.so.3 (0xb7f4c000)
```

> **Tip**
>
> Mac OS X does not have `ldd`, but you can obtain the equivalent functionality by running `otool` with option `-L`.

It is quite possible to have a configuration in which Apache uses its bundled PCRE and ModSecurity another PCRE version available on the system. This is probably the most commonly reported issue on the users' mailing list because it not only prevents ModSecurity from working but also crashes Apache.

Starting with version 2.6, ModSecurity reports at startup (in the error log) the detected library version numbers and compares them to those used at compile time. One or more warnings will be issued if a mismatch is found. This feature is especially handy for troubleshooting PCRE library collisions.

```
[notice] ModSecurity for Apache/2.6.0-rc1 (http://www.modsecurity.org/) configured.
[notice] ModSecurity: APR compiled version="1.3.8"; loaded version="1.3.8"
[notice] ModSecurity: PCRE compiled version="8.12"; loaded version="8.12 2011-01-15"
[notice] ModSecurity: LIBXML compiled version="2.7.3"
```

Installation from Binaries

As previously discussed, using a binary version of ModSecurity is often the easiest option, because it just works. Unfortunately, what you gain in ease of installation you lose by sometimes being able to use only an older version. Further, packagers often do not include `mlogc`, which is essential for remote log centralization. For that reason, I don't recommend installation from binaries unless there is no other way. If you do decide to check out what is available in binary form, the list of available binary packages is available from http://www.modsecurity.org/download/.

Fedora Core, CentOS, and Red Hat Enterprise Linux

If you are a Fedora Core user and you are running version 10 or later, you can install ModSecurity directly from the official distribution, using `yum`:

```
# yum install mod_security
```

On CentOS and Red Hat Enterprise Linux, you have two options. One is to use the packages from EPEL (Extra Packages for Enterprise Linux), which is a volunteer effort and part of the Fedora community. The other option is to use the custom packages built by Jason Litka. Either way, the available packages support CentOS/RHEL 4.x and 5.x, on the i386 and x86_64 architectures. The installation process is the same as for Fedora Core.

Debian and Ubuntu

Debian was the first distribution to include ModSecurity, but also the first distribution to kick it out due to a licensing issue. The ModSecurity license was clarified in version 2.5.6 with the addition of an open source exception, and that opened the door for it to get back into Debian. Alberto Gonzalez Iniesta has been a long-time supporter of ModSecurity on Debian, supporting ModSecurity in his own (unofficial) repository, and is now the official packager.

If you are running a version of Debian or Ubuntu that includes ModSecurity, the installation is as easy as:

```
# apt-get install libapache-mod-security
```

This single command will download the package and install it, then activate the module in the Apache configuration.

> **Note**
>
> Don't forget that Debian uses a special system to manage Apache modules and sites. To activate and deactivate modules, use a2enmod and a2dismod, respectively. To manage Apache, use apache2ctl.

Installation on Windows

ModSecurity was ported to Windows early on, in 2003, and has run well on this platform ever since. Windows binary packages of ModSecurity are maintained by Steffen Land, who runs Apache Lounge, a community for those who run Apache on Windows. In addition to ModSecurity, Steffen maintains his version of Apache itself, as well as many third-party modules you might want to run on Windows. The ModSecurity binary packages are consistently up to date, so you will have little trouble if you want to run the latest version. The download includes ModSecurity as well as mlogc.

> **Note**
>
> Although it might be possible to run Steffen's ModSecurity binaries with a version of Apache produced elsewhere, you should really use only the packages from a single

> location that are intended to be used together. If you don't, you may encounter unusual behavior and web server crashes.

The installation is quite easy. First, download the package and copy the dynamic libraries into the `modules/` folder (of the Apache installation). Then, modify your Apache configuration to activate ModSecurity:

```
LoadModule security2_module modules/mod_security2.so
```

You will also need to activate `mod_unique_id`. This module may not be already active, but there should already be a commented-out line in your configuration. You just need to find it and uncomment it. If it isn't there, just add the following:

```
LoadModule unique_id_module modules/mod_unique_id.so
```

Summary

It's never been easier to install ModSecurity, now that it is included with so many operating system and distributions. Although the installation from source code gives you guaranteed access to the most recent version, as well as access to the yet-unreleased code, it can be time-consuming if you are not used to it. It's not everyone's cup of tea. There is something to be said for using the provided version and not having to think about upgrading (and saving the time it takes to upgrade).

In the next chapter, I'll explain each of the configuration options, teaching you how to set every single option, step by step, so that everything is just the way you like it.

3 Configuration

Now that you have ModSecurity compiled and ready to run, we can proceed to the configuration. This section, with its many subsections, goes through every part of ModSecurity configuration, explicitly configuring every little detail:

- Going through all the configuration directives will give you a better understanding of how ModSecurity works. Even if there are features that you don't need immediately, you will learn that they exist and you'll be able to take advantage of them when the need arises.

- By explicitly configuring every single feature, you will foolproof your configuration against incompatible changes to default settings that may happen in future versions of ModSecurity.

In accordance with its philosophy, ModSecurity won't do anything implicitly. It won't even run unless you tell it to. There are three reasons for that:

1. By not doing anything implicitly, we ensure that ModSecurity does only what you tell it to. That not only keeps you in control, but it also makes you think about every feature before you add it to your configuration.

2. It is impossible to design a default configuration that works in all circumstances. We can give you a framework within which you can work (as I am doing in this section), but you still need to shape your configuration according to your needs.

3. Security is not free. You pay for it by the increased consumption of RAM, CPU, or the possibility that you may block a legitimate request. Incorrect configuration may cause problems, so we need you to think carefully about what you're doing.

The remainder of this section explains the proposed default configuration for ModSecurity. You can get a good overview of the default configuration simply by examining the configuration directives supported by ModSecurity, which are listed in Table 3.1, "Main configuration directives" (with the exception of the logging directives, which are listed in several tables in the Chapter 4, *Logging*).

Table 3.1. Main configuration directives

Directive	Description
SecArgumentSeparator	Sets the application/x-www-form-urlencoded parameter separator
SecCookieFormat	Sets the cookie parser version
SecDataDir	Sets the folder for persistent storage
SecRequestBodyAccess	Controls request body buffering
SecRequestBodyInMemoryLimit	Sets the size of the per-request memory buffer
SecRequestBodyLimit	Sets the maximum request body size ModSecurity will accept
SecRequestBodyLimitAction	Controls what happens once the request body limit is reached
SecRequestBodyNoFilesLimit	Sets the maximum request body size, excluding uploaded files
SecResponseBodyAccess	Controls response body buffering
SecResponseBodyLimit	Specifies the response body buffering limit
SecResponseBodyLimitAction	Controls what happens once the response body limit is reached
SecResponseBodyMimeType	Specifies a list of response body MIME types to inspect
SecResponseBodyMimeTypesClear	Clears the list of response body MIME types
SecRuleEngine	Controls the operation of the rule engine
SecTmpDir	Sets the folder for temporary files

Folder Locations

Your first configuration task is to decide where on the filesystem to put the various bits and pieces that every ModSecurity installation consists of. Installation layout is often a matter of taste, so it is difficult for me to give you advice. Similarly, different choices may be appropriate in different circumstances. For example, if you are adding ModSecurity to a web server and you intend to use it only occasionally, you may not want to use an elaborate folder structure, in which case you'll probably put the ModSecurity folder underneath Apache's. When you're using ModSecurity as part of a dedicated reverse proxy installation, however, a well–thought out structure is something that will save you a lot of time in the long run.

I prefer to always use an elaborate folder layout, because I like things to be neat and tidy, and because the consistency helps me when I am managing multiple ModSecurity installations. I start by creating a dedicated folder for ModSecurity (`/opt/modsecurity`) with multiple sub-folders underneath. The subfolders that are written to at runtime are all grouped (in `/opt/modsecurity/var`), which makes it easy to relocate them to a different filesystem using a symbolic link. I end up with the following structure:

Binaries
  `/opt/modsecurity/bin`

Configuration files
  `/opt/modsecurity/etc`

Audit logs
    `/opt/modsecurity/var/audit`

Persistent data
    `/opt/modsecurity/var/data`

Logs
    `/opt/modsecurity/var/log`

Temporary files
    `/opt/modsecurity/var/tmp`

File uploads
    `/opt/modsecurity/var/upload`

Getting the permissions right may involve slightly more effort, depending on your circumstances. Most Apache installations bind to privileged ports (e.g., 80 and 443), which means that the web server must be started as `root`, and that further means that `root` must be the principal owner of the installation. Because it's not good practice to stay `root` at runtime, Apache will switch to a low-privilege account (we'll assume it's apache) as soon as it initializes. You'll find the proposed permissions in Table 3.2, "Folder permissions".

Table 3.2. Folder permissions

Location	Owner	Group	Permissions
`/opt/modsecurity`	root	apache	`rwxr-x---`
`/opt/modsecurity/bin`	root	apache	`rwxr-x---`
`/opt/modsecurity/etc`	root	root	`rwx------`
`/opt/modsecurity/var`	root	apache	`rwxr-x---`
`/opt/modsecurity/var/audit`	apache	root	`rwx------`
`/opt/modsecurity/var/data`	apache	root	`rwx------`
`/opt/modsecurity/var/log`	root	root	`rwx------`
`/opt/modsecurity/var/tmp`	apache	apache	`rwxr-x---`
`/opt/modsecurity/var/upload`	apache	root	`rwx------`

I've arrived at the desired permission layout through the following requirements:

1. As already discussed, it is `root` that owns everything by default, and we assign ownership to `apache` only where that is necessary.

2. In two cases (`/opt/modsecurity` and `/opt/modsecurity/var`), we need to allow `apache` to access a folder so that it can get to a subfolder; we do this by creating a group, also

called `apache`, of which user `apache` is the only member. We use the same group for the `/opt/modsecurity/bin` folder, where you might store some binaries Apache will need to execute at runtime.

3. One folder, `/opt/modsecurity/var/log`, stands out, because it is the only folder underneath `/opt/modsecurity/var` where `apache` is not allowed to write. That folder contains log files that are opened by Apache early on, while it is still running as `root`. On any Unix system, you *must* have only one account with write access to that folder, and it has to be the principal owner. In our case, that must be `root`. Doing otherwise would create a security hole, whereby the `apache` user would be able to obtain partial `root` privileges using symlink trickery. (Essentially, in place of a log file, the `apache` user creates a symlink to some other `root`-owned file on the system. When Apache starts it runs as `root` and opens for writing the system file that the `apache` user would otherwise be unable to touch. By submitting requests to Apache, one might be able to control exactly what is written to the log files. That can lead to system compromise.)

4. A careful observer will notice that I've allowed group folder access to `/opt/modsecurity/var/tmp` (which means that any member of the `apache` group is allowed to read the files in the folder) even though this folder is owned by `apache`, which already has full access. This is because you will sometimes want to allow ModSecurity to exchange information with a third user account—for example, if you want to scan uploaded files for viruses (usually done using ClamAV). To allow the third user account to access the files created by ModSecurity, you just need to make it a member of the `apache` group and relax the file permissions using the `SecUploadFileMode` directive.

> **Note**
>
> As an exception to the proposed layout, you may want to reuse Apache's log directory for ModSecurity logs. If you don't, you'll have the error log separate from the debug log (and the audit log if you choose to use the serial logging format). In a reverse proxy installation in particular, it makes great sense to keep everything integrated and easier to find. There may be other good reasons for breaking convention. For example, if you have more than one hard disk installed and you use the audit logging feature a lot, you may want to split the I/O operations across the disks.

Configuration Layout

If you have anything but a trivial setup, spreading configuration across several files is necessary in order to make maintenance easier. There are several ways to do that, and some have more to do with taste than anything else, but in this section I will describe an approach that is good enough to start with.

Whatever configuration design I use, there is usually one main entry point, typically named `modsecurity.conf`, which I use as a bridge between Apache and ModSecurity. In my bridge

file, I refer to any other ModSecurity files I might have, such as those listed in Table 3.3, "Configuration files".

Table 3.3. Configuration files

Filename	Description
main.conf	Main configuration file
rules-first.conf	Rules that need to run first
rules.conf	Your principal rule file
rules-last.conf	Rules that need to run last

Your main configuration file (`modsecurity.conf`) may thus contain only the following lines:

```
<IfModule mod_security2.c>
Include /opt/modsecurity/etc/main.conf
Include /opt/modsecurity/etc/rules-first.conf
Include /opt/modsecurity/etc/rules.conf
Include /opt/modsecurity/etc/rules-last.conf
</IfModule>
```

The `<IfModule>` tag is there to ensure that the ModSecurity configuration files are used only if ModSecurity is active in the web server. This is common practice when configuring any nonessential Apache modules and allows you to deactivate a module simply by commenting out the appropriate `LoadModule` line.

Adding ModSecurity to Apache

As the first step, make Apache aware of ModSecurity, adding the needed components. Depending on how you've chosen to run ModSecurity, this may translate to adding one or more lines to your configuration file. This is what the lines may look like:

```
# Load libxml2
LoadFile /usr/lib/libxml2.so
# Load Lua
LoadFile /usr/lib/liblua5.1.so
# Finally, load ModSecurity
LoadModule security2_module modules/mod_security2.so
```

Now you just need to tell Apache where to find the configuration:

```
Include /opt/modsecurity/etc/modsecurity.conf
```

Powering Up

ModSecurity has a master switch—the `SecRuleEngine` directive—that allows you to quickly turn it on and off. This directive will always come first in every configuration. I generally

recommend that you start in detection-only mode, because that way you are sure nothing will be blocked.

```
# Enable ModSecurity, attaching it to every transaction.
SecRuleEngine DetectionOnly
```

You will normally want to keep this setting enabled, of course, but there will be cases in which you won't be exactly sure whether ModSecurity is doing something it shouldn't be. Whenever that happens, you will want to set it to Off, just for a moment or two, until you perform a request without it running.

The SecRuleEngine directive is context-sensitive (i.e., it works with Apache's container tags <VirtualHost>, <Location>, and so on), which means that you are able to control exactly where ModSecurity runs. You can use this feature to enable ModSecurity only for some sites, parts of a web site, or even for a single script only. I discuss this feature in detail later.

Will ModSecurity Block in Detection-Only Mode?

You might be expecting ModSecurity to never block when you configure SecRuleEngine with DetectionOnly, but that behavior is only the case for version 2.6 and later; earlier releases may behave otherwise. First of all, you should be aware that there are two cases in which Mod-Security will block without being instructed to do so by a rule, even in detection mode. If ModSecurity sees more request data than it is configured to handle, it will respond with a HTTP_REQUEST_ENTITY_TOO_LARGE (413) error code. The three directives that control the request buffer sizes are: SecRequestLimit, SecRequestNoFilesLimit, and SecResponseBodyLimit.

Prior to ModSecurity 2.5, the only option was to block when the limit was reached. As of version 2.5, you can use SecResponseBodyLimitAction and change how ModSecurity behaves when the response body limit is reached. If you choose ProcessPartial, ModSecurity will stop accepting response body data, but it will not block. Starting with version 2.6, you can use SecRequestBodyLimit action, which has a similar effect on request body processing. When running in blocking mode, the default setting for both directives is Reject; in detection-only mode, the default setting is ProcessPartial.

Request Body Handling

Requests consist of two parts: the headers part, which is always present, and the body, which is optional. Use the SecRequestBodyAccess directive to tell ModSecurity to look at request bodies:

```
# Allow ModSecurity to access request bodies. If you don't,
# ModSecurity won't be able to see any POST parameters
# and that's generally not what you want.
```

```
SecRequestBodyAccess On
```

Once this feature is enabled, ModSecurity will not only have access to the content transmitted in request bodies, but it will also completely buffer them. The buffering is essential for reliable attack prevention. With buffering in place, your rules have the opportunity to inspect requests in their entirety, and only after you choose not to block will the requests be allowed through.

The downside of buffering is that, in most cases, it uses RAM for storage, which needs to be taken into account when ModSecurity is running embedded in a web server. There are three directives that control how buffering is done. The first two, `SecRequestBodyLimit` and `SecRequestBodyNoFilesLimit`, establish request limits:

```
# Maximum request body size we will accept for buffering.
# If you support file uploads then the value given on the
# first line has to be as large as the largest file you
# want to accept. The second value refers to the size of
# data, with files excluded. You want to keep that value
# as low as practical.
SecRequestBodyLimit 1310720
SecRequestBodyNoFilesLimit 131072
```

In the versions prior to 2.5, ModSecurity supported only `SecRequestBodyLimit` (which establishes an absolute limit on a request body), but that directive turned out to be impractical in combination with file uploads. File uploads generally do not use RAM (and thus do not create an opportunity for a denial of service attack), which means that it is safe to allow such large requests. Unfortunately, doing so also meant allowing large requests that are not file uploads, defying the purpose for which the directive was introduced in the first place. The second directive, `SecRequestBodyNoFilesLimit`, which was introduced in ModSecurity 2.5, calculates request body sizes slightly differently, ignoring the space taken up by files. In practice, this means that the maximum allowed request body size will be that specified in the `SecRequestBodyNoFilesLimit` directive, with the exception of file uploads, where the setting in `SecRequestBodyLimit` takes precedence.

> **Warning**
>
> When the directive `SecStreamInBodyInspection` is enabled, it will attempt to store the entire raw request body in `STREAM_INPUT_BODY`. In this case, you lose the protection of `SecRequestBodyNoFilesLimit`; the maximum amount of memory consumed for buffering will be that defined with `SecRequestBodyLimit`.

> **Note**
>
> In blocking mode, ModSecurity will respond with a 413 (Request Entity Too Large) response status code when a request body limit is reached. This response code was chosen to mimic what Apache does in similar circumstances. See `SecRequestBodyLimitAction` for more information.

The third directive that deals with buffering, SecRequestBodyInMemoryLimit, controls how much of a request body will be stored in RAM, but it only works with file upload (multipart/form-data) requests:

```
# Store up to 128 KB of request body data in memory. When
# the multipart parser reaches this limit, it will start
# using your hard disk for storage. That is generally slow,
# but unavoidable.
SecRequestBodyInMemoryLimit 131072
```

The request bodies that fit within the limit configured with SecRequestBodyInMemoryLimit will be stored in RAM. The request bodies that are larger will be streamed to disk. This directive allows you to trade performance (storing request bodies in RAM is fast) for size (the storage capacity of your hard disk is much bigger than that of your RAM).

Response Body Handling

Similarly to requests, responses consist of headers and a body. Unlike requests, however, most responses have bodies. Use the SecResponseBodyAccess directive to tell ModSecurity to observe (and buffer) response bodies:

```
# Allow ModSecurity to access response bodies. We leave
# this disabled because most deployments want to focus on
# the incoming threats, and leaving this off reduces
# memory consumption.
SecResponseBodyAccess Off
```

I prefer to start with this setting disabled, because many deployments don't care to look at what leaves their web servers. Keeping this feature disabled means ModSecurity will use less RAM and less CPU. If you care about output, however, just change the directive setting to On.

There is a complication with response bodies, because you generally only want to look at the bodies of some of the responses. Response bodies make the bulk of the traffic on most web sites, and the majority of that are just static files that don't have any security relevance in most cases. The response MIME type is used to distinguish the interesting responses from the ones that are not. The SecResponseBodyMimeType directive lists the response MIME types you are interested in.

```
# Which response MIME types do you want to look at? You
# should adjust the configuration below to catch documents
# but avoid static files (e.g., images and archives).
SecResponseBodyMimeType text/plain text/html
```

You can control the size of a response body buffer using the SecResponseBodyLimit directive:

```
# Buffer response bodies of up to 512 KB in length.
SecResponseBodyLimit 524288
```

The problem with limiting the size of a response body buffer is that it breaks sites whose pages are longer than the limit. In ModSecurity 2.5, we introduced the SecResponseBodyLimitAction directive, which allows ModSecurity users to choose what happens when the limit is reached:

```
# What happens when we encounter a response body larger
# than the configured limit? By default, we process what
# we have and let the rest through.
SecResponseBodyLimitAction ProcessPartial
```

If the setting is Reject, the response will be discarded and the transaction interrupted with a 500 (Internal Server Error) response code. If the setting is ProcessPartial, which I recommend, ModSecurity will process what it has in the buffer and allow the rest through.

At the first thought, it may seem that allowing the processing of partial response bodies creates a security issue. For the attacker who controls output, it seems easy to create a response that is long enough to bypass observation by ModSecurity. This is true. However, if you have an attacker with full control of output, it is impossible for any type of monitoring to work reliably. For example, such an attacker could encrypt output, in which case it will be opaque to ModSecurity. Response body monitoring works best to detect information leakage, configuration errors, traces of attacks (successful or not), and data leakage in the cases when an attacker does not have full control of output.

Other than that, response monitoring is most useful when it comes to preventing the data leakage that comes from low-level error messages (e.g., database problems). Because such messages typically appear near the beginning of a page, the ProcessPartial setting will work just as well to catch them.

Filesystem Locations

We've made the decisions regarding filesystem locations already, so all we need to do now is translate them into configuration. The following two directives tell ModSecurity where to create temporary files (`SecTmpDir`) and where to store persistent data (`SecDataDir`):

```
# The location where ModSecurity will store temporary files
# (for example, when it needs to handle a multipart request
# body that is larger than the configured limit). If you don't
# specify a location here your system's default will be used.
# It is recommended that you specify a location that's private.
SecTmpDir /opt/modsecurity/var/tmp/

# The location where ModSecurity will keep its data. This,
# too, needs to be a path that other users can't access.
SecDataDir /opt/modsecurity/var/data/
```

File Uploads

Next, we configure the handling of file uploads. We configure the folder where ModSecurity will store intercepted files, but we keep this functionality disabled for now. File upload interception slows down ModSecurity and can potentially consume a lot of disk space, so you'll want to enable this functionality only in the places where you really need it.

```
# The location where ModSecurity will store intercepted
# uploaded files. This location must be private to ModSecurity.
SecUploadDir /opt/modsecurity/var/upload/

# By default, do not intercept (nor store) uploaded files.
SecUploadKeepFiles Off
```

For now, we also assume that you will not be using external scripts to inspect uploaded files. That allows us to keep the file permissions more secure, by allowing access only to the apache user:

```
# Uploaded files are by default created with permissions that
# do not allow any other user to access them. You may need to
# relax that if you want to interface ModSecurity to an
# external program (e.g., an anti-virus).
SecUploadFileMode 0600
```

> **Note**
>
> You may experience permission-related issues in ModSecurity versions 2.5.10 and 2.5.11 (but not in 2.5.12 and better), in which process umask affects the permissions of newly created files. Thus, if you set SecUploadFileMode to 0660, but the umask is 022, the resulting permissions will be 0644 (0660 & ~022). If you need to change the process umask, edit the script that you use to control Apache (e.g., apachectl in my case). For example, assuming that you wish to set the umask to 002, add umask 002 to the end of the script.

If you are using ModSecurity 2.5.12 or later, you should lower the maximum number of files that ModSecurity will handle in a request:

```
# Limit the number of files we are willing
# to handle in any one request.
SecUploadFileLimit 32
```

The default value is 100, but that's usually too much. The issue here is that it is very easy for an attacker to include many embedded files in a single multipart/form-data request (for example, hundreds and even thousands), but also that you don't want ModSecurity to create that many files on the filesystem (which happens only if the storage or validation of uploaded files is required), because it would create a denial of service situation.

Debug Log

Debug logging is very useful for troubleshooting, but in production you want to keep it at minimum, because too much logging will affect the performance. The recommended debug log level for production is 3, which will duplicate in the debug log what you will also see in Apache's error log. This is handy, because the error log will grow at a faster rate and may be rotated. A copy of the ModSecurity messages in the debug log means that you always have all the data you need.

```
# Debug log
SecDebugLog /opt/modsecurity/var/log/debug.log
SecDebugLogLevel 3
```

Audit Log

In ModSecurity terminology, "audit logging" refers to the ability to record complete transaction data. For a typical transaction without a request body, this translates to roughly 1 KB. Multiply that by the number of requests you are receiving daily and you'll soon realize that you want to keep this type of logging to an absolute minimum.

Our default configuration will use audit logging only for the transactions that are *relevant*, which means those that have had an error or a warning reported against them. Other possible values for SecAuditEngine are On (log everything) and Off (log nothing).

```
# Log only what is really necessary.
SecAuditEngine RelevantOnly
```

In addition, we will also log the transactions with response status codes that indicate a server error (500–599). You should never see such transactions on an error-free server. The extra data logged by ModSecurity may help you uncover security issues, or problems of some other type.

```
# Also log requests that cause a server error.
SecAuditLogRelevantStatus ^5
```

By default, we log all transaction data except response bodies. This assumes that you will seldom log (as it should be), because response bodies can take up a lot of space.

```
# Log everything we know about a transaction.
SecAuditLogParts ABDEFHIJKZ
```

Using the same assumption, we choose to use a single file to store all the recorded information. This is not adequate for the installations that will log a lot and prevents remote logging, but it is good enough to start with:

```
# Use a single file for logging.
SecAuditLogType Serial
SecAuditLog /opt/modsecurity/var/log/audit.log
```

As the final step, we will configure the path that will be used in the more scalable audit logging scheme, called *concurrent logging*, even though you won't need to use it just yet:

```
# Specify the path for concurrent audit logging.
SecAuditLogStorageDir /opt/modsecurity/var/audit/
```

Miscellaneous Options

The directives covered in this section are seldom needed, but having them will allow us to achieve complete coverage of the ModSecurity configuration options. You'll also be aware that they exist and will be able to use them in the rare cases where they are needed.

The `SecArgumentSeparator` directive allows you to change the parameter separator used for the `application/x-www-form-urlencoded` encoding, which is used to transport all `GET` parameters and most `POST` parameters.

```
SecArgumentSeparator &
```

Virtually all applications use an ampersand for this purpose, but some may not. The HTML 4.01 specification recommends that applications support the use of semicolons as separators (see section *B.2.2 Ampersands in URI attribute values*) for convenience. In PHP, for example, it is possible to use any character as a separator.

The `SecCookieFormat` directive selects one of the two cookie parsers available in ModSecurity. Virtually all applications use Netscape-style cookies (sometimes also known as version 0 cookies), so there will be little reason to change this setting:

```
SecCookieFormat 0
```

Default Rule Match Policy

As we're nearing the end of the configuration, you need to decide what you want to happen when a rule matches. It is recommended that you start without blocking, because that will allow you to monitor the operation of your installation over a period of time and ensure that legitimate traffic is not being marked as suspicious:

```
SecDefaultAction "phase:1,log,auditlog,pass"
```

This default policy will work for all rules that follow it in the same configuration context.

> **Note**
>
> It is possible to write rules that ignore the default policies. If you are using third-party rule sets and you are not sure how they will behave, consider switching the entire engine to detection only (using `SecRuleEngine`). No rule will block when you do that, regardless of how it was designed to work.

Handling Processing Errors

As you may recall from our earlier discussion, ModSecurity avoids making decisions for you. It will detect problems as they occur, but it will generally leave to you to deal with them. In our default configuration, we will have a couple of rules to deal with the situations that ModSecurity can't deal with on its own—processing errors.

There are currently three types of processing errors:

- Request and response buffering limits encountered
- Parsing errors
- PCRE limit errors

Normally, you don't need to be too concerned about encountering buffer limits, because they often occur during normal operation. If you do want to take them into account when making decisions, you can use the INBOUND_DATA_ERROR and OUTBOUND_DATA_ERROR variables for request and response buffering, respectively.

ModSecurity parsers are designed to be as permissive as possible without compromising security. They will raise flags when they fail, but also when they encounter something suspicious. By checking the flags in your rules you detect the processing errors.

Currently the only parsing errors that can happen are the request body processor errors. We will use two rules to handle those.

The first rule will examine the REQBODY_PROCESSOR_ERROR flag for errors. This flag will be raised whenever a request body parsing error occurs, regardless of which parser was used for parsing:

```
# Verify that we've correctly processed the request body.
# As a rule of thumb, when failing to process a request body
# you should reject the request (when deployed in blocking mode)
# or log a high-severity alert (when deployed in detection-only mode).
SecRule REQBODY_PROCESSOR_ERROR "!@eq 0" \
    "phase:2,t:none,log,block,msg:'Failed to parse request body: ↵
%{REQBODY_PROCESSOR_ERROR_MSG}'"
```

The second rule is specific to the multipart/form-data parser, which is used to handle file uploads. If it detects a problem, it produces an error message detailing the flaws:

```
# By default be strict with what you accept in the multipart/form-data
# request body. If the rule below proves to be too strict for your
# environment consider changing it to detection-only. You are encouraged
# _not_ to remove it altogether.
SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"phase:2,t:none,log,block,msg:'Multipart request body \
failed strict validation: \
PE %{REQBODY_PROCESSOR_ERROR}, \
BQ %{MULTIPART_BOUNDARY_QUOTED}, \
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
```

```
DB %{MULTIPART_DATA_BEFORE}, \
DA %{MULTIPART_DATA_AFTER}, \
HF %{MULTIPART_HEADER_FOLDING}, \
LF %{MULTIPART_LF_LINE}, \
SM %{MULTIPART_MISSING_SEMICOLON}, \
IQ %{MULTIPART_INVALID_QUOTING}, \
IF %{MULTIPART_INVALID_HEADER_FOLDING}, \
FE %{MULTIPART_FILE_LIMIT_EXCEEDED}'"
```

Errors specific to the multipart parsers should never occur unless an attacker genuinely tries to bypass ModSecurity by manipulating the request body payload. Some versions of ModSecurity did have false positives in this area, but the most recent version should be false-positive-free. If you do encounter such a problem, feel free to post it to the mod-security-users mailing list. It will mean that you've encountered an interesting attacker or a ModSecurity bug.

PCRE limit errors are available starting with ModSecurity version 2.5.12, when the way PCRE is used was changed to significantly lower the match and recursion limits. SecPcreMatchLimit and SecPcreMatchLimitRecursion were also added to allow users to control these settings. The lower the PCRE limits, the more difficult it is to subvert PCRE and the regular expressions to commit denial of service attacks. As of ModSecurity version 2.5.13, the debug log will identify rules that have exceeded the limits. For example:

```
[3] Rule 9905c50 [id "-"][file "/home/ivanr/apache/conf/httpd.conf"][line "525"] - ↵
Execution error - PCRE limits exceeded (-8): (null).
```

We will leave the PCRE limits defaults as they are, but add a rule to warn us when they are exceeded:

```
SecRule TX:MSC_PCRE_LIMITS_EXCEEDED "@eq 1" \
    "phase:5,t:none,log,pass,msg:'PCRE limits exceeded'"
```

I've used phase 5 for the rule, but if you are really paranoid and think that exceeding PCRE limits is grounds for blocking, switch to phase 2 (and change `pass` to something else).

Verifying Installation

After you're done installing and configuring ModSecurity, it is recommended that you undertake a short exercise to ensure everything is in order:

1. Add a simple blocking rule to detect something in a parameter. For example, the following rule will inspect all parameters for the word MY_UNIQUE_TEST_STRING, responding with a 503 (*Service Unavailable*) on a match:

```
SecRule ARGS MY_UNIQUE_TEST_STRING \
    "phase:2,log,deny,status:503"
```

2. Restart Apache, using the `graceful` restart option if your server is in production and you don't want any downtime.

3. Send a `GET` request, using your browser, to the ModSecurity-protected server, including the "attack payload" in a parameter (i.e., `http://www.example.com/?test=MY_UNIQUE_TEST_STRING`). ModSecurity should block the request.

4. Verify that the message has appeared in both the error log and the debug log, and that the audit log contains the complete transaction.

5. Submit a `POST` request that triggers the test rule. With this request, you are testing whether ModSecurity will see the request body, and whether it will be able to pass the data in it to your backend after inspection. For this test, in particular, it is important that you're testing with the actual application you want to protect. Only doing so will exercise the entire stack of components that make the application. This test is important because of the way Apache modules are written (very little documentation, so module authors generally employ any approach that "works" for them)—you can generally never be 100% certain that a third-party module was implemented correctly. For example, it is possible to write a module that will essentially hijack a request early on and bypass all the other modules, including ModSecurity. We are doing this test simply because we don't want to leave anything to chance.

6. If you want to be really pedantic (I have been, on many occasion—you can never be too sure), you may want to consider writing a special test script for your application, which will somehow record the fact that it has been invoked (mine usually writes to a file in `/tmp`). By sending a request that includes an attack—which will be intercepted by ModSecurity—and verifying that the script has not been invoked, you can be completely sure that blocking works as intended.

7. Remove the test rule and restart Apache again.

8. Finally, and just to be absolutely sure, examine the permissions on all Apache and ModSecurity locations and verify that they are correct.

You're done!

Summary

In this chapter, we took the time to look at every configuration option of ModSecurity. Strictly speaking, we could have left many of the options at their defaults and spent about a tenth of this time on configuration. But I've always found it better to explicitly define every setting, because with that approach, you end up with the configuration that's tailored to your needs. In addition to that, you get to know ModSecurity better, which might prove crucial at some point in the future.

We didn't pay much attention to logging in this chapter, opting to configure both the debug log and the audit log very conservatively. But there's a wealth of logging options in ModSecurity. In the next chapter, I'll discuss logging in detail, and conclude with the configuration topics.

4 Logging

This section covers the logging capabilities of ModSecurity in detail. Logging is a big part of what ModSecurity does, so it will not surprise you to learn that there are extensive facilities available for your use.

Debug Log

The debug log is going to be your primary troubleshooting tool, especially initially, while you're learning how ModSecurity works. You are likely to spend a lot of time with the debug log cranked up to level 9, observing why certain things work the way they do. There are two debug log directives, as you can see in Table 4.1, "Debug log directives".

Table 4.1. Debug log directives

Directive	Description
SecDebugLog	Path to the debug log file
SecDebugLogLevel	Debug log level

In theory, there are 10 debug log levels, but not all are used. You'll find the ones that are in Table 4.2, "Debug log levels". Messages with levels 1–3 are designed to be meaningful, and are copied to the Apache's error log. The higher-level messages are there mostly for troubleshooting and debugging.

Table 4.2. Debug log levels

Debug log level	Description
0	No logging
1	Errors (e.g., fatal processing errors, blocked transactions)
2	Warnings (e.g., non-blocking rule matches)
3	Notices (e.g., non-fatal processing errors)
4	Informational
5	Detailed
9	Everything!

You will want to keep the debug log level in production low (either at 3 if you want a copy of all messages in the debug log, or at 0 if you're happy with having them only in the error log). This is because you can expect in excess of 50 debug log messages (each message is an I/O operation) and at least 7 KB of data for an average transaction. Logging all that for every single transaction consumes a lot of resources.

This is what a single debug log line looks like:

```
[18/Aug/2009:08:18:08 +0100] [192.168.3.111/sid#80f4e40][rid#81d0588][/index.html]↵
[4] Initialising transaction (txid SopVsH8AAAEAAE8-NB4AAAAD).
```

The line starts with metadata that is often longer than the message itself: the time, client's IP address, internal server ID, internal request ID, request URI, and finally, the debug log level. The rest of the line is occupied by the message, which is essentially free-form. You will find many examples of debug log messages throughout this guide, which I've used to document how ModSecurity works.

Debugging in Production

There's another reason for avoiding extensive debug logging in production, and that's simply that it's very difficult. There's usually so much data that it sometimes takes you ages to find the messages pertaining to the transaction you wish to investigate. In spite of the difficulties, you may occasionally need to debug in production because you can't reproduce a problem elsewhere.

> **Note**
>
> ModSecurity 2.5 extended the audit logging functionality by being able to record in the audit log all the rules that matched. This feature is very helpful, as it minimizes the need for debugging in production, but it still can't tell you why some rules *didn't* match.

One way to make debugging easier is to keep debug logging disabled by default and enable it only for the part of the site that you wish to debug. You can do this by overriding the default configuration using the `<Location>` context directive. While you're doing that, it may be a good idea to specify a different debug log file altogether. That way you'll keep main debug log file free of your tests.

```
<Location /myapp/>
    SecDebugLogLevel 9
    SecDebugLog /opt/modsecurity/var/log/troubleshooting.log
</Location>
```

This approach, although handy, still does not guarantee that the volume of information in the debug log will be manageable. What you really want is to enable debug logging for the requests you send. ModSecurity provides a solution for this by allowing a debug log level to be changed at runtime, on a per-request basis. This is done using the special `ctl` action that allows some of the configuration to be updated at runtime.

All you need to do is somehow uniquely identify yourself. In some circumstances, observing the IP address will be sufficient:

```
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,nolog,pass,ctl:debugLogLevel=9
```

Using your IP address won't work in the cases when you're hidden by a NAT of some sort, and share an IP address with a bunch of other users. One straightforward approach is to modify your browser settings to put a unique identifier in your `User-Agent` request header. (How exactly that is done depends on the browser you are using. In Firefox, for example, you can add a general.useragent.override setting to your configuration, or use one of the many extensions specifically designed for this purpose.)

```
SecRule REQUEST_HEADERS:User-Agent YOUR_UNIQUE_ID \
    phase:1,nolog,pass,ctl:debugLogLevel=9
```

This approach, although easy, has a drawback: all your requests will cause an increase in debug logging. You may think of an application in terms of dynamic pages, but extensive debug logging will be enabled for every single embedded object, too. Also, if you're dealing with an application that you're using frequently, you may want to avoid excessive logging.

The most accurate way of dynamically enabling detailed debug logging is to manually indicate, to ModSecurity, the exact requests on which you want it to increase logging. You can do this by modifying your `User-Agent` string on request-by-request basis, using one of the tools that support request interception and modification. (The Tamper Data extension does that for Firefox.) Armed with such a tool, you submit your requests in your browser, modify them in the tool, and then allow them through modified. It's a bit involved, but a time-saver overall. And, while you are at it, it is a good idea to make your identifiers similar enough for your rule

to always detect them, but different enough to allow you to use a search function to quickly find the exact request in a file with thousands.

Audit Log

It is a little-known fact that I originally started to work on ModSecurity because I was frustrated with not being able to log full HTTP transaction data. The audit log, which does just that, was one of the first features implemented.

Table 4.3. Audit log directives

Directive	Description
SecAuditEngine	Controls the audit log engine; possible values On, Off, or RelevantOnly
SecAuditLog	Path to an audit log file
SecAuditLog2	Path to another audit log file (copy)
SecAuditLogParts	Specifies which part of a transaction will be logged
SecAuditLogRelevantStatus	Specifies which response statuses will be considered relevant
SecAuditLogStorageDir	Path there concurrent audit log files will be stored
SecAuditLogType	Specifies the type of audit log to use: Serial or Concurrent

A typical audit log entry (short, GET request without a body and no logging of the response body) consumes around 1.3 KB. Requests with bodies will increase the amount of data that needs to be logged, as well as the logging of response bodies.

Logically, each audit log entry is a single file. When serial audit logging is used, all entries will be placed within one file, but with concurrent audit logging, one file per entry is used. Looking at a single audit log entry, you'll find that it consists of multiple independent segments (parts):

```
--6b253045-A--
...
--6b253045-B--
...
--6b253045-C--
...
--6b253045-F--
...
--6b253045-E--
...
--6b253045-H--
...
--6b253045-Z--
```

A segment begins with a boundary and ends when the next segment begins. The only exception is the terminating segment (Z), which consists only of the boundary. The idea behind the use of multiple segments is to allow each audit log entry to contain potentially different information. Only the parts A and Z are mandatory; the use of the other parts is controlled with the SecAuditLogParts directive. Table 4.4, "Audit log parts" contains the list of all audit log parts, along with a description of their purpose.

Table 4.4. Audit log parts

Part letter	Description
A	Audit log header (mandatory)
B	Request headers
C	Request body
D	Reserved
E	Response body
F	Response headers
G	Reserved
H	Audit log trailer, which contains additional data
I	Compact request body alternative (to part C), which excludes files
J	Information on uploaded files (available as of version 2.6.0)
K	Contains a list of all rules that matched for the transaction
Z	Final boundary (mandatory)

Audit Log Entry Example

Every audit log entry begins with part A, which contains the basic information about the transaction: time, unique ID, source IP address, source port, destination IP address, and destination port:

```
--6b253045-A--
[18/Aug/2009:08:25:15 +0100] SopXW38EAAE9YbLQ 192.168.3.1 2387 192.168.3.111 8080
```

Part B contains the request headers and nothing else:

```
--6b253045-B--
POST /index.html?a=test HTTP/1.1
Host: 192.168.3.111:8080
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
```

```
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.3.111:8080/index.html?a=test
Content-Type: application/x-www-form-urlencoded
Content-Length: 6
```

Part C contains the raw request body, typically that of a POST request:

```
--6b253045-C--
b=test
```

Part F contains the response headers:

```
--6b253045-F--
HTTP/1.1 200 OK
Last-Modified: Tue, 18 Aug 2009 07:17:44 GMT
ETag: "6eccf-99-4716550995f20"
Accept-Ranges: bytes
Content-Length: 159
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Part E contains the response body:

```
--6b253045-E--
<html><body><h1>It works!</h1></body></html>

<form action="index.html?a=test" method="POST">
<textarea name="b">test</textarea>
<input type=submit>
</form>
```

The final part, H, contains additional transaction information.

```
--6b253045-H--
Stopwatch: 1250580315933960 1902 (551* 580 978)
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.5.9 (http://www.modsecurity.org/).
Server: Apache/2.2.11 (Unix) DAV/2
```

Part K contains a list of rules that matched in a transaction. It is not unusual for this part to be empty, but if you have a complex rule set, it may show quite a few rules. Audit logs that record transactions on which there were warnings, or those that were blocked, will contain at least one rule here. In this example you'll find a rule that emits a warning on every request:

```
--6b253045-K--
```

```
SecAction "phase:2,auditlog,log,pass,msg:'Matching test'"
```

Every audit log file ends with the terminating boundary, which is part Z:

```
--6b253045-Z--
```

Concurrent Audit Log

Initially, ModSecurity supported only the serial audit logging format. Concurrent logging was introduced to address two issues:

- Serial logging is only adequate for moderate use, because only one audit log entry can be written at any one time. Serial logging is fast (logs are written at the end of every transaction, all in one go) but it does not scale well. In the extreme, a web server performing full transaction logging practically processes only one request at any one time.

- Real-time audit log centralization requires individual audit log entries to be deleted once they are dealt with, which is impossible to do when all alerts are stored in a single file.

Concurrent audit logging changes the operation of ModSecurity in two aspects. To observe the changes, switch to concurrent logging without activating `mlogc` by changing `SecAuditLogType` to `Concurrent` (don't forget to restart Apache).

First, as expected, each audit log entry will be stored in a separate file. The files will not be created directly in the folder specified by `SecAuditLogStorageDir`, but in an elaborate structure of subfolders whose names will be constructed from the current date and time:

```
./20090822
./20090822/20090822-1324
./20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQAAAAAA
./20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA
```

The purpose of the scheme is to prevent too many files from being created within one directory; many filesystems have limits that can be relatively quickly reached on a busy web server. The first two parts in each filename are based on time (`YYYYMMDD` and `HHMMSS`). The third parameter is the unique transaction ID.

In addition to each entry getting its own file, the format of the main audit log file will change when concurrent logging is activated. The file that previously stored the entries themselves will now be used as a record of all generated audit log files.

```
192.168.3.130 192.168.3.1 - - [22/Aug/2009:13:24:20 +0100] "GET / HTTP/1.1" 200 ↵
56 "-" "-" SojdH8AAQEAAAugAQAAAAAA "-" ↵
/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQAAAAAA 0 1248 ↵
md5:8b097f4f880852e179e7b63b68a7fc92
192.168.3.130 192.168.3.1 - - [22/Aug/2009:13:24:20 +0100] "GET /favicon.ico ↵
```

```
HTTP/1.1" 404 267 "-" "-" SojdH8AAQEAAAugAQEAAAAA "-" ↵
/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA 0 1226 ↵
md5:c76740f076a3cb759d62fb610ab39342
```

The index file is similar in principle to a web server access log. Each line describes one transaction, duplicating some of the information already available in audit log entries. The purpose of the index file is two-fold:

- The first part, which duplicates some of the information available in audit logs, serves as a record of everything that you have recorded so that you can easily search through it.

- The second part tells you where an audit log entry is stored (e.g., `/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA`), where it begins within that file (always zero, because this feature is not used), how long it is, and gives you its MD5 hash (useful to verify integrity).

When real-time audit log centralization is used, this information is not written to a file. Instead, it is written to a pipe, which means that it is sent directly to another process, which deals with the information immediately. You will see how that works in the next section.

Remote Logging

ModSecurity comes with a tool called `mlogc` (short for ModSecurity Log Collector), which can be used to transport audit logs in real time to a remote logging server. This tool has the following characteristics:

Secure
> The communication path between your ModSecurity sensors and the remote logging server is secured with SSL and authenticated using HTTP Basic Authentication.

Efficient
> Remote logging is implemented with multiple threads of execution, which allow for many alerts to be handled in parallel. Existing HTTP connections are reused.

Reliable
> An audit log entry will be deleted from the sensor only once its safe receipt is acknowledged by the logging server.

Buffered
> The `mlogc` tool maintains its own audit entry queue, which has two benefits. First, if the logging server is not available the entries will be preserved, and submitted once the server comes back online. Second, `mlogc` controls the rate at which audit log entries are submitted, meaning that a burst of activity on a sensor will not result in an uncontrolled burst of activity on the remote logging server.

Note

Remote logging uses a simple but very effective protocol based on HTTP. You'll find it documented in Chapter 20, *Data Formats*.

If you've followed my installation instructions, you will have `mlogc` compiled and sitting in your `bin/` folder. To proceed, you will need to configure it, then add it to the ModSecurity configuration.

How Remote Logging Works

Remote logging in ModSecurity is implemented through an elaborate scheme designed to minimize the possibility of data loss. Here is how it's done:

1. ModSecurity processes a transaction and creates an audit log entry file on disk, as explained in the section called "Concurrent Audit Log".

2. ModSecurity then notifies the `mlogc` tool, which runs in a separate process. The notification contains enough information to locate the audit log entry file on disk.

3. The `mlogc` tool adds the audit log entry information to the in-memory queue and to its transaction log (file `mlogc-transaction.log` by default).

4. One of many worker threads that run within `mlogc` takes the audit log entry and submits it to a remote logging server. The entry is then removed from the in-memory queue and the transaction log is notified.

5. A periodic checkpoint operation, initiated by `mlogc`, writes the in-memory queue to the disk (file `mlogc-queue.log` by default) and erases the transaction log.

If `mlogc` crashes, Apache will restart it automatically. When an unclean shutdown is detected, `mlogc` will reconstruct the entry queue using the last known good point (the on-disk queue) and the record of all events since the moment the on-disk queue was created, which are stored in the transaction log.

Configuring Remote Logging

The `mlogc` configuration file is similar to that of Apache, only simpler. First we need to tell `mlogc` where its "home" is, which is where it will create its log files. Log files are very important, because—as it is Apache that starts `mlogc` and ModSecurity that talks to it—we never interact with `mlogc` directly. We'll need to look in the log files for clues in case of problems.

```
# Specify the folder where the logs will be created
CollectorRoot  /opt/modsecurity/var/log

# Define what the log files will be called. You probably
```

```
# won't ever change the names, but mlogc requires you
# to define it.
ErrorLog        mlogc-error.log

# The error log level is a number between 0 and 5, with
# level 3 recommended for production (5 for troubleshooting).
ErrorLogLevel 3

# Specify the names of the data files. Similar comment as
# above: you won't want to change these, but they are required.
TransactionLog mlogc-transaction.log
QueuePath       mlogc-queue.log
LockFile        mlogc.lck
```

Then we tell it where to find audit log entries. The value given to LogStorageDir should be the same as the one you provided to ModSecurity's SecAuditLogStorageDir:

```
# Where are the audit log entries created by ModSecurity?
LogStorageDir /opt/modsecurity/var/audit
```

Next, we need to tell mlogc where to submit the audit log entries. We identify a remote server with a URL and credentials:

```
# Remote logging server details.
ConsoleURI "https://REMOTE_ADDRESS:8888/rpc/auditLogReceiver"
SensorUsername "USERNAME"
SensorPassword "PASSWORD"
```

The remaining configuration directives aren't required, but it's always a good idea to explicitly configure your programs, rather than let them use their defaults:

```
# How many parallel connections to use to talk to the server,
# and how much to wait (in milliseconds) between submissions.
# These two directives are used to control the rate at which
# audit log entries are submitted.
MaxConnections      10
TransactionDelay    50

# How many entries is a single thread allowed to process
# before it must shut down.
MaxWorkerRequests   1000

# How long to wait at startup before really starting.
StartupDelay        5000

# Checkpoints are periods when the entries from the transaction
# log (which is written to sequentially) are consolidated with
# the entries in the main queue.
```

```
CheckpointInterval  15

# Back-off time after goes away or responds with a fatal error.
ServerErrorTimeout  60
```

> **Note**
>
> The `mlogc` tool will take audit log entries created by ModSecurity, submit them to a
> remote logging server and delete them from disk, but it will leave the empty folders
> (that were used to store the entries) behind. You will have to remove them yourself,
> either manually or with a script.

Activating Remote Logging

You will need to make two changes to your default configuration. First you need to switch to
concurrent audit logging, because that's the only way `mlogc` can work:

```
SecAuditLogType Concurrent
```

Next you need to activate `mlogc`, which is done using the piped logging feature of Apache:

```
SecAuditLog "|/opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf"
```

The pipe character at the beginning of the line tells Apache to treat what follows as a command
line. As a result, whenever you start Apache from now on, it will start a copy of `mlogc` in turn,
and keep it running in parallel, leaving a one-way communication channel that will be used
by ModSecurity to inform `mlogc` of every new audit log entry it creates.

Your complete configuration should look like this now:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^5
SecAuditLogParts ABDEFHIJKZ
SecAuditLogType Concurrent
SecAuditLog "|/opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf"
SecAuditLogStorageDir /opt/modsecurity/var/audit/
```

If you restart Apache now, you should see `mlogc` running:

```
USER      PID  COMMAND
root     11845  /usr/sbin/apache2 -k start
root     11846  /opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf
apache   11847  /usr/sbin/apache2 -k start
apache   11848  /usr/sbin/apache2 -k start
apache   11849  /usr/sbin/apache2 -k start
apache   11850  /usr/sbin/apache2 -k start
apache   11851  /usr/sbin/apache2 -k start
```

If you go to the log/ folder, you should see two new log files:

```
dev:/opt/modsecurity/var/log# l
total 1684
drwx------ 2 root root      4096 2009-08-20 10:31 .
drwxr-x--- 7 root apache    4096 2009-08-18 20:01 ..
-rw-r----- 1 root root    926530 2009-08-20 08:09 audit.log
-rw-r----- 1 root root    771903 2009-08-20 08:09 debug.log
-rw-r--r-- 1 root root       696 2009-08-20 10:33 mlogc-error.log
-rw-r--r-- 1 root root         0 2009-08-20 10:31 mlogc-transaction.log
```

If you look at the mlogc-error.log file, there will be signs of minimal activity (the timestamps from the beginning of every line were removed for clarity):

```
[3] [11893/0] Configuring ModSecurity Audit Log Collector 2.5.10-dev2.
[3] [11893/0] Delaying execution for 5000ms.
[3] [11895/0] Configuring ModSecurity Audit Log Collector 2.5.10-dev2.
[3] [11895/0] Delaying execution for 5000ms.
[3] [11893/0] Queue file not found. New one will be created.
[3] [11893/0] Caught SIGTERM, shutting down.
[3] [11893/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating normally.
[3] [11895/0] Queue file not found. New one will be created.
```

It is normal for two copies of mlogc to have run, because that's how Apache treats all piped logging programs. It starts two (one while it's checking configuration), but leaves only one running. The second token on every line in the example is the combination of process ID and thread ID. Thus you can see how there are two processes running at the same time (PID 11893 and PID 11895). Because only one program can handle the data files, mlogc is designed to wait for a while before it does anything. Basically, if it still lives after the delay, that means it's the copy that's meant to do something.

What happens if you make an error in the configuration file, which is preventing mlogc from working properly? As previously discussed, mlogc can't just respond to you on the command line, so it will do the only thing it can. It will report the problem and shut down. (Don't be surprised if Apache continues with attempts to start it. That's what Apache does with piped logging programs.)

If you make a mistake in defining the error log, you may actually get an error message in response to the attempt to start Apache. Following is the error message you'd get if you left ErrorLog undefined:

```
dev:/opt/modsecurity/etc# apache2ctl start
[1] [12026/0] Failed to open the error log (null): Bad address
[3] [12026/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating with error 1
```

If mlogc managed to open its error log, it will do the expected and write all error messages there. For example:

```
[1] [11985/0] QueuePath not defined in the configuration file.
[3] [11985/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating with error 1
```

At this point, it is a good idea to delete the serial audit log file audit.log, or store it elsewhere. Having switched to concurrent logging, that file won't be updated anymore and it will only confuse you.

Troubleshooting Remote Logging

Assuming default logging configuration (level 3), a single audit log entry handled by mlogc will produce one line in the log file:

```
[3] [2435/693078] Entry completed (0.080 seconds, 9927 bytes): ↵
SsHOykMXI18AAAmnIgAAAABC
```

That's basically all you need to know—that an entry has been safely transmitted to the intended destination. The status codes 200 and 409 are taken to mean that an entry has been successfully processed. The response code 200 indicates that there were no problems with the entry; the response code 409 indicates that the entry is faulty, but that it has been accepted by the server (and that mlogc should delete it and move on).

You will get more information when something gets wrong, of course. For example, you will see the following message whenever your logging server goes down:

```
[2] [2435/693078] Flagging server as errored after failure to submit entry ↵
SsHPNOMXI18AAAmLHucAAAAG (cURL code 7): couldn't connect to host
```

The message will appear on the first failed delivery, and then once every minute until the server becomes operational. This is because mlogc will shut down its operation for a short period whenever something unusual happens with the server. Only one thread of operation will continue to work to probe the server, with processing returning to full speed once the server recovers. You'll get the following information in the log:

```
[3] [2435/693078] Clearing the server error flag after successful entry ↵
submission: SsHPNOMXI18AAAmLHucAAAAG
[3] [2435/693078] Entry completed (0.684 seconds, 9927 bytes): ↵
SsHPNOMXI18AAAmLHucAAAAG
```

Going back to the error message, the first part tells you that there's a problem with the server; the second part tells you what the problem is. In the previous case, the problem was "couldn't connect to host", which means the server is down.

Table 4.5. Common remote logging problems

Error message	Description
couldn't connect to host	The server could not be reached. It probably means that the server itself is down, but it could also indicate a network issue. You can investigate the cURL return code to determine the exact cause of the problem.
Possible SSL negotiation error	Most commonly, this message will mean that you configured `mlogc` to submit over plain-text, but the remote server uses SSL. Make sure the `ConsoleURI` parameter starts with "https://".
Unauthorized	The credentials are incorrect. Check the `SensorUsername` and `SensorPassword` parameters.
For input string: "0, 0"	A remote server can indicate an internal error using any response status code other than 200 and 409, but such errors are treated as transient. ModSecurity Community Console has a long-standing problem where it responds with a 500 code to an audit log entry that is invalid in some way. The use of the 500 response code makes `mlogc` pause and attempt to deliver again, only to see the Console fail again. The process continues indefinitely and the only solution at the moment is to track down the offending audit log entry on the sensor and manually delete it.

If you still can't resolve the problem, I suggest that you increase the `mlogc` error log level from 3 (`NOTICE`) to 5 (`DEBUG2`), restart Apache (graceful will do), and try to uncover more information that would point to a solution. Actually, I advise you to perform this exercise even before you encounter a problem, because an analysis of the detailed log output will give you a better understanding of how `mlogc` works.

File Upload Interception

File upload interception is a special form of logging, in which the files being uploaded to your server are intercepted, inspected, and stored, and all that before they are seen by an application. The directives related to this feature are in Table 4.6, "File upload directives", but you've already seen them all in the section called "File Uploads".

Table 4.6. File upload directives

Directive	Description
`SecUploadDir`	Specifies the location where intercepted files will be stored
`SecUploadFileMode`	Specifies the permissions that will be used for the stored files
`SecUploadKeepFiles`	Specifies whether to store the uploaded files (`On`, `Off`, or `RelevantOnly`)

Storing Files

Assuming the default configuration suggested in this guide, you only need to change the setting of the `SecUploadKeepFiles` directive to `On` to start collecting uploaded files. If, after a few

file upload requests, you examine /opt/modsecurity/var/upload, you'll find files with names similar to these:

```
20090818-164623-SorMz38AAAEAAFG2AOAAAAAA-file-ok0c4T
20090818-164713-SorNAX8AAAEAAFG4AbUAAAAC-file-2ef1eC
```

You can probably tell that the first two parts of a filename are based on the time of upload, then follows the unique transaction ID, the -file- part that is always the same, and a random string of characters at the end. ModSecurity uses this algorithm to generate file names primarily to avoid filename collision and support the storage of a large number of files in a folder. In addition, avoiding the use of a user-supplied file name prevents a potential attacker from placing a file with a known name on a server.

When you store a file like this, it is just a file and it doesn't tell you anything about the attacker. Thus, for the files to be useful, you also need to preserve the corresponding audit log entries, which will contain the rest of the information.

> **Note**
>
> Storage of intercepted files can potentially consume a lot of disk space. If you're doing it, you should at least ensure that the filesystem that you're using for storage is not the root filesystem—you don't want an overflow to kill your entire server.

Inspecting Files

For most people, a more reasonable SecUploadKeepFiles setting is RelevantOnly, which enables the storage of only the files that have failed inspection in some way. For this setting to make sense, you need to have at least one external inspection script along with a rule that invokes it.

A file inspection rule is rather simple:

```
SecRule FILES_TMPNAMES "@inspectFile /opt/modsecurity/bin/file-inspect.pl" \
    phase:2,t:none,log,block
```

This example rule will invoke the script /opt/modsecurity/bin/file-inspect.pl for every uploaded file. The script will be given the location of the temporary file as its first and only parameter. It can do whatever it wants with the contents of the file, but it is expected to return a single line of output that consists of a verdict (1 if everything is in order and 0 for a fault), followed by an error message. For example:

```
1 OK
```

Or:

```
0 Error
```

Following are the debug log lines produced by the inspection file:

```
[4] Recipe: Invoking rule 99e6538; [file "/opt/modsecurity/etc/rules.conf"] ↵
[line "3"].
[5] Rule 99e6538: SecRule "FILES_TMPNAMES" "@inspectFile ↵
/opt/modsecurity/bin/file-inspect.pl" "phase:2,auditlog,t:none,log,block"
[4] Transformation completed in 2 usec.
[4] Executing operator "inspectFile" with param ↵
"/opt/modsecurity/bin/file-inspect.pl" against FILES_TMPNAMES:f.
[9] Target value: ↵
"/opt/modsecurity/var/tmp//20090819-181304-SowyoH8AAQEAACW1AIo-file-ZPtFAq"
[4] Executing /opt/modsecurity/bin/file-inspect.pl to inspect ↵
/opt/modsecurity/var/tmp//20090819-181304-SowyoH8AAQEAACW1AIoAAAAA-file-ZPtFAq.
[9] Exec: /opt/modsecurity/bin/file-inspect.pl
[4] Exec: First line from script output: "1 OK"
[4] Operator completed in 6969 usec.
[4] Rule returned 0.
```

If an error occurs, for example if you make a mistake in the name of the script, you'll get an error message that looks similar to this one:

```
[9] Exec: /opt/modsecurity/bin/file_inspect.pl
[1] Exec: Execution failed while reading output: ↵
/opt/modsecurity/bin/file_inspect.pl (End of file found)
```

> **Tip**
>
> If you write your inspection scripts in Lua, ModSecurity will be able to execute them directly using an internal Lua engine. The internal execution will not only be faster, but from the Lua scripts you will be able to access the complete transaction context (which is not available to any external programs).

Integrating with ClamAV

ClamAV (*http://www.clamav.net*) is a popular open source anti-virus program. If you have it installed, the following script will allow you to utilize it to scan files from ModSecurity:

```perl
#!/usr/bin/perl

$CLAMSCAN = "/usr/bin/clamscan";

if (@ARGV != 1) {
    print "Usage: modsec-clamscan.pl FILENAME\n";
    exit;
}

my ($FILE) = @ARGV;
```

```perl
$cmd = "$CLAMSCAN --stdout --disable-summary $FILE";
$input = `$cmd`;
$input =~ m/^(.+)/;
$error_message = $1;

$output = "0 Unable to parse clamscan output";

if ($error_message =~ m/: Empty file\.$/) {
    $output = "1 empty file";
}
elsif ($error_message =~ m/: (.+) ERROR$/) {
    $output = "0 clamscan: $1";
}
elsif ($error_message =~ m/: (.+) FOUND$/) {
    $output = "0 clamscan: $1";
}
elsif ($error_message =~ m/: OK$/) {
    $output = "1 clamscan: OK";
}

print "$output\n";
```

> **Note**
>
> If you need a file to test with, you can download one from *http://www.eicar.org/
> anti_virus_test_file.htm*. The files at this location contain a test signature that will be
> picked up by ClamAV.

The error message from the integration script will return either the result of the inspection of the file or an error message if the inspection process failed. Following is an example that shows a successful detection of a "virus":

```
[9] Exec: /opt/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamscan: Eicar-Test-Signature"
[4] Operator completed in 2137466 usec.
[2] Warning. File ↵
"/opt/modsecurity/var/tmp//20090819-181833-Sowz6X8AAQEAACXQAWAAAAAB-file-logg59" ↵
rejected by the approver script "/opt/modsecurity/bin/modsec-clamscan.pl": 0 ↵
clamscan: Eicar-Test-Signature [file "/opt/modsecurity/etc/rules.conf"] [line "3"]
```

If you look carefully at the example output, you'll see that the inspection took in excess of 2 seconds. This is not unusual (even for my slow virtual server), because we're creating a new instance of the ClamAV engine for every inspection. The scanning alone is fast, but the initialization takes considerable time. A more efficient method would be to use the ClamAV daemon (e.g., the `clamav-daemon` package on Debian) for inspection. In this case, the daemon is running all the time, and the script is only informing it that it needs to inspect a file.

Assuming you've followed the recommendation for the file permissions settings given in the section called "Folder Locations", this is what you need to do:

1. Change the name of the ClamAV script from `clamscan` to `clamdscan` (note the added `d` in the filename).

2. Add the ClamAV user (typically `clamav`) to the group `apache`.

3. Relax the default file permissions used for uploaded files to allow group read, by changing `SecUploadFileMode` from `0600` to `0640`.

An examination of the logs after the change in the configuration will tell you that there's been a significant improvement—from seconds to milliseconds:

```
[9] Exec: /opt/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamscan: Eicar-Test-Signature"
[4] Operator completed in 13329 usec.
[2] Warning. File ↵
"/opt/modsecurity/var/tmp//20090819-182404-Sow1NH8AAQEAACiEAIcAAAAA-file-AMzbgK" ↵
rejected by the approver script "/opt/modsecurity/bin/modsec-clamscan.pl": 0 ↵
clamscan: Eicar-Test-Signature [file "/opt/modsecurity/etc/rules.conf"] [line "3"]
```

Advanced Logging Configuration

By now you have seen how you have many facilities you can use to configure logging to work exactly as you need it. The facilities can be grouped into four categories:

Static logging configuration
The various audit logging configuration directives establish the default (or static) audit logging configuration. You should use this type of configuration to establish what you want to happen in most cases. You should then use the remaining configuration mechanisms (listed next) to create exceptions to handle edge cases.

Setting of the relevant flag on rule matches
Every rule match, unless suppressed, increments the transaction's *relevant* flag. This handy feature, designed to work with the `RelevantOnly` setting of `SecAuditEngine`, allows you to trigger transaction logging when something unusual happens.

Per-rule logging suggestions
Rule matching and the actions `auditlog` and `noauditlog` do not control logging directly. They should be viewed as mere suggestions—it is up to the engine to decide whether to log a transaction. They are also ephemeral, affecting only the rules with which they are associated. They will be forgotten as the processing moves onto the next rule.

Dynamic logging configuration
Rules can make logging decisions that affect entire decisions (through the `ctl` action), but that functionality should not be used lightly. Most rules should be concerned only

with event generation, restricting their decisions to the suggestions mentioned in the previous. The ability to affect transaction logging should be used by system rules placed in phase 5 and written specifically for the purpose of logging control.

Increasing Logging from a Rule

Using the `SecAuditLogParts` directive, you will configure exactly what parts (how much information) you want logged for every transaction, but one setting will not be adequate in all cases. For example, most configurations will not be logging response bodies, but that information is often required to determine whether certain types of attack (XSS, for example) were successful.

The following rule will detect only simple XSS attacks, but when it does, it will cause the transaction's response body to be recorded:

```
SecRule ARGS <script> phase:2,log,block,ctl:auditLogParts=+E
```

Dynamically Altering Logging Configuration

The feature discussed in the previous section is very useful, but you may not always like the fact that some rules are changing what you're logging. I know I would not. Luckily, it's a problem that can be resolved with an addition of a phase 5 rule that resets the logged audit log parts:

```
SecAction phase:5,nolog,pass,ctl:auditLogParts=ABCDFGH
```

You can then decide on your own whether the logging of part E is justified. If you are using full audit logging in particular, you will need to manually increase the amount you log per transaction. The `HIGHEST_SEVERITY` variable, which contains the highest severity of the rules that matched during a transaction, is particularly useful:

```
SecRule HIGHEST_SEVERITY "@le 2" phase:5,nolog,pass,ctl:auditLogParts=+E
```

Removing Sensitive Data from Audit Logs

Most web application programmers are taught to always use POST methods for the transactions that contain sensitive data. After all it is well known that request bodies are never logged, meaning that the sensitive data will never be logged, either. ModSecurity changes this situation, because it allows for full transaction logging. To deal with the sensitive data that may find its way into the logs, ModSecurity uses the sanitation actions `sanitiseArg`, `sanitiseRequestHeader` and `sanitiseResponseHeader`, and `sanitiseMatched`. You basically just need to tell ModSecurity which elements of a transaction you want removed and it will remove them for you, replacing their values in the log with asterisks. The first three actions all require parameters that you will typically know at configuration time, which means that you will in-

voke them unconditionally with SecAction. Sanitation works when invoked from any phase, but you should always use phase 5, which is designed for this type of activity.

Use sanitiseArg to prevent the logging of the parameters whose names you know. For example, let's assume that you have an application that uses the parameters password, oldPassword, and newPassword to transmit, well, passwords. This is what you'll do:

```
SecAction "phase:5,nolog,pass,\
    sanitiseArg:password,\
    sanitiseArg:oldPassword,\
    sanitiseArg:newPassword"
```

Similarly, use sanitiseRequestHeader and sanitiseResponseHeader to remove the contents of the headers whose names you know. For example, if you have an application that uses HTTP Basic Authentication, you will need the following rule to prevent the passwords from being logged:

```
SecAction "phase:5,nolog,pass,\
    sanitiseRequestHeader:Authorization"
```

The last action, sanitiseMatched, is used when you need to sanitize a parameter whose name you don't know in advance. My first example will sanitize the contents of every parameter that has the word password in the name:

```
SecRule ARGS_NAMES password "phase:5,nolog,pass,\
    sanitiseMatched"
```

In the following example, we look for anything that resembles a credit card number and then sanitize it:

```
SecRule ARGS "@verifyCC \d{13,16}" "phase:5,nolog,pass,\
    sanitiseMatched"
```

Finally, the sanitiseMatchedBytes action, available as of ModSecurity version 2.6.0, can remove only the parts of input that contain sensitive information and not entire parameters. This action works only in conjunction with operators that are based on regular expressions (e.g., @rx, @verifyCC, etc.), and further requires the capture action to be specified.

```
SecRule ARGS "@verifyCC \d{13,16}" "phase:5,nolog,pass,\
    capture,sanitiseMatchedBytes"
```

When further parameters are specified, this new operator can even leave parts of the sensitive data in the log. In the following example, we leave the first four digits and the last four digits of a credit card number:

```
SecRule ARGS "@verifyCC \d{13,16}" "phase:5,nolog,pass,\
    capture,sanitiseMatchedBytes:4/4"
```

Selective Audit Logging

Although full transaction logging (the logging of every single byte of every single transaction) sounds good in theory, in practice it is very difficult to use, because it slows down your sever and requires huge amounts of storage space. There are ways to get some of the same benefits for a fraction of cost by using partial full logging on demand.

The trick is to tie in logging to the tracking of IP addresses, users, or sessions. By default you will log only what is relevant, but when you spot something suspicious coming from (for example) an IP address, you may change your logging configuration to turn on full logging for the offending IP address only. Here's how.

First you need to set up IP address tracking. You do this only once for all your rules, so it should usually be part of your main configuration:

```
SecAction phase:1,nolog,pass,initcol:ip=%{REMOTE_ADDR}
```

Now you need to add a phase rule that will trigger logging when something else happens. In the following case, we want to start logging everything coming from an IP address after a single rule match. To achieve that, we set the flag `ip.logflag` for up to one hour (3600 seconds):

```
SecRule HIGHEST_SEVERITY "@gt 0" \
    phase:5,nolog,pass,setvar:ip.logflag=1,expirevar:ip.logflag=3600
```

Finally, we add a rule that detects the flag and forces logging:

```
SecRule IP:logflag "@gt 0" \
    phase:5,nolog,pass,ctl:auditLogEngine=On
```

Summary

This chapter, along with the two before it, covered the configuration of ModSecurity. You learned how to install ModSecurity and how to configure it, with special attention given to the logging facilities. Logging deserved its own chapter, because configuring a tool to perform certain actions is often only half of the entire story, with the other half consisting of tracking exactly what happened and why. Further, remote logging is a gateway to other systems, which may assist you in managing ModSecurity.

The next three chapters discuss a new topic: rule writing. You'll first read an overview of the entire rule language, followed by a tutorial in rule writing and then by a higher-level discussion on how to place ModSecurity configuration within Apache's own directives. Finally, the interesting bits are here!

Index