SECOND EDITION

MODSECURITY HANDBOOK

The Complete Guide to the Popular Open Source Web Application Firewall

Free edition: Getting Started

Christian Folini
Ivan Ristić

Feisty Duck

Last update: Mon Nov 29 12:55:10 GMT 2021 (build 23)

ModSecurity Handbook

Christian Folini, Ivan Ristić

ModSecurity Handbook: Getting Started

by Christian Folini, Ivan Ristić

ISBN: 978-1-90711708-4

Published in July 2017 (build 23). First edition published in March 2010.

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: Melinda Rankin

Cover design: Peter Jovanović

Cover illustration: Maja Veselinović

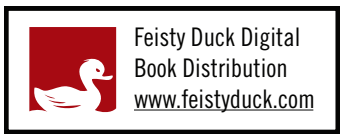

Table of Contents

Preface to the Free Edition

Welcome to *ModSecurity Handbook: Getting Started*. This is a special free version that consists of the first four chapters of the full book. Since *ModSecurity Handbook* went into print, we 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 who 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 to the Second Edition

I was an early adopter of ModSecurity. I first came across it in about 2005 and was immediately intrigued. Here was a tool that could help me improve my life, indirectly, by improving the security of the systems I manage. I started to use it, although you could say it grew on me slowly over time. You see, I'm a medievalist who landed in web server security when my application for a job at an open air museum was declined. In parallel with the new job, I was also running one of the better known reenactment companies, recreating medieval life for a museum audience. I got married, we started a family, we purchased a historical house (with all the strings attached), and though ModSecurity became more and more important to me over the years, it remained a day job because evenings and weekends were already occupied.

I slowly started to teach Apache and ModSecurity courses, I published blog posts and tutorials on the use of ModSecurity, and, last year, the day job started to expand into evenings and weekends when I became involved with the OWASP ModSecurity Core Rule Set project. I joined a very active team and became invested in the development of the Core Rules Paranoia Mode and Sampling Mode, two core features of the Core Rule Set 3.0 release.

When Ivan asked me to write this new edition of *ModSecurity Handbook*, it felt like a culmination of my work with ModSecurity! This work allowed me to explore features and areas I hadn't used before. It gave me a better view of ModSecurity and—shhh, don't tell anyone—I am probably the person who profited most from it.

I started with an overhaul of the reference section of the book. About one-third of it is brand-new, because many new features were added to ModSecurity in the six years since the first edition of this book was published. Another significant effort was adding more detail throughout and many examples to better explain what each feature did. This is especially visible with the transformations that now come with handy "before" and "after" examples, which provide much-needed clarity about exactly how data is changed. The idea behind this expansion was to describe the usage of the software in a consistent way and to give people who know the online reference substantial value when they buy the book.

The prose part of the book saw fewer updates: some additions to most chapters, small fixes here and there, rewordings, and removing legacy explanations or historical information (e.g.,

new features in version 2.5.12). All in all, I blew away the dust from that part of the book. This is not true for Chapter 10, *Performance*, which was updated with substantial new data obtained from many different test runs in multiple scenarios. This allowed me to assess performance anew, and I was able to show that the performance of ModSecurity transformations now is not quite how it was when the first edition was written (now it's better!).

You don't write a book on your own, and you don't get into a position to write a technical book on your own, either. Many people contributed in their own way to my work, and I can only name a few of them here. Let their names stand in for many more people like them. First and foremost, my thanks belong to my company of many years, netnea in Berne, Switzerland. Netnea's decision to hire a PhD (me, still hot from the press and hitherto specialized in German mysticism) allowed me to start on this adventure in the first place.

Jelena Girić-Ristić from Feisty Duck, this book's publisher, accompanied me from the moment I accepted this project, and her good spirit kept me working during days when gray clouds covered the sky. Ivan, who wrote the first edition of this book, acted as a technical editor this time around and offered his guiding hand to help achieve clarity when explaining complex topics. Osama Elnaggar, Walter Hop, Marco Pizzoli, and Chaim Sanders reviewed the manuscript and pointed out shortcomings that I had overlooked. Finally, Melinda Rankin came in as copyeditor when they were done and gave the manuscript a most welcome polish.

My marvelous wife Saara is the rare sort of a pastor running a Linux desktop and helping her techie husband configure his mobile phone. She put up with me when I grew grumpy or felt lost with all this book writing, and she cheered me up with her understanding. Our two boys had to put up with me as well, and I feel sorry for all the playing and fun that we missed out on in recent months.

But in the end, what really made this book possible was my experience working with Apache and ModSecurity over the years, in turn possible thanks to my customers, who trusted my growing knowledge and who placed their security projects into my hands. I can't name them all, of course, but I will name Swiss Post, my most important customer. The management at Swiss Post allowed me and the engineering team to invest into a carefully designed reverse proxy platform we are all very proud of. This success was of primordial importance to this book. Other customers bring new challenges with every project, and they all teach me new concepts and new ways to run Apache and ModSecurity. It's a great adventure every day.

Preface to the First Edition

I didn't mean to write this book; I really didn't. In late 2008 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'm thrilled to say. The package is complete. You have an open source tool able to compete with the best commercial products out there, *and* you have the documentation to match.

With this book, I'm also trying something completely new: *continuous writing and publishing*. You see, I 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, meaning that even when you get the paper version (as most people seem to prefer to), it's 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's as simple as that. ModSecurity is a fantastic tool, but it's 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'll find that every new chapter advances a notch. Even if you're a long-time ModSecurity user, I believe you'll benefit from a fresh start. I'll 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.

However, there's 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 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 move 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*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 insight into how it works, and it finishes with an overview of the key areas you'll 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'll find useful in your day-to-day work.

Chapter 2, *Installation*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*Chapter 3, *Configuration*, explains how each of the available configuration directives should be used. By the end of the chapter, you'll have a complete overview of the configuration options and a solid default configuration for all your ModSecurity installations.

Chapter 4, *Logging*Chapter 4, *Logging*, addresses 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 sanitization feature to prevent sensitive data from being stored in the logs.

Chapter 5, *Rule Language Overview*, is the first of three chapters that address rule writing. This chapter contains an overview of the entire rule language, which will get you started and provide 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 fun chapter that adopts a gradual approach, introducing features one by one. By the end of the chapter, you'll 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'll first learn how to put a few rules together and add them to the configuration, then learn 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 whitelist-

ing, virtual patching, IP address reputation, and blacklisting. You'll 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 and moving 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 rulesets, discussing how to use the rulesets others have written and how to write your own.

Chapter 10, *Performance*, covers several performance-related topics. It opens with an overview of how 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 rulesets in order to give you a taste of what they're like, as well as to document a methodology you can use to test your own rules. The chapter then moves to ruleset benchmarking, which is an essential part of the process of rule writing. The last part of this chapter gives 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 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 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's especially useful when integrating with external systems.

Chapter 13, *Handling XML*, covers the XML capabilities of ModSecurity in detail. You'll 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'll 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,[1] 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.

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 interactions among its author(s) and its 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's no more waiting for years to see 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).

[1] Feisty Duck web site (Feisty Duck, retrieved 29 Dec 2016)

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: ModSecurity is a tool that will help you sleep better at night; in this book, we'll explain how. We usually call ModSecurity a *web application firewall* (WAF), the generally accepted term to refer to the class of products designed specifically to secure web applications. Other times, we call it an *HTTP intrusion detection tool*, because we think that name better describes what ModSecurity does. Neither name is entirely adequate, but we don't have a better one. However, it doesn't really matter what we call it. The point is that web applications—yours, ours, everyone's—are terribly insecure on average. We all struggle to keep ahead of security issues and need any help we can get to handle them.

Ivan thought to create ModSecurity while he was responsible for the security of several web-based products. He could see how insecure most web applications were, slapped together with little time spent on design and even less time spent on understanding security issues. Not only were web applications insecure, but people generally had little awareness of whether they were being attacked or exploited. Most web servers kept only standard 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 can see HTTP traffic, you can 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 its source code.

Brief History of ModSecurity

Like many other open source projects, ModSecurity started out as a hobby. Back in 2002, producing secure web applications was virtually impossible. (It's the same these days, sadly.) However, that realization led to the idea of a tool that would sit in front of web applications and control the flow of data to and from the system. 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 ModSecurity, and its popularity started to rise.

Initially, most development effort for the tool went into wrestling with Apache to make request body inspection possible. Apache 1.3.x didn't include any interception or filtering APIs, but it was still possible to trick it into submission. Apache 2.x improved the situation by providing APIs that allowed content interception, but no documentation was available.

By 2004, Ivan converted from obsessing about software development to obsessing about web application security. He quit his job and started treating ModSecurity as a business. In the summer of 2006, ModSecurity went head-to-head with other web application firewalls in an evaluation conducted by Forrester Research, and it achieved great results. Later that year, ModSecurity was acquired by Breach Security. A team of one eventually became a team of many: Brian Rectanus came to work on ModSecurity, Ofer Shezaf embarked on the rules, and Ryan C. Barnett handled community management and education. ModSecurity 2.0, a complete rewrite, was released in late 2006. Breach Security also released ModSecurity Community Console, which combined the functionality of a remote logging sensor and a monitoring and reporting GUI.

In 2009, Ivan left Breach Security. He stayed involved with ModSecurity for a while, but mostly worked on the first edition of this book. In his own words, he couldn't leave the project if it wasn't properly documented. Brian Rectanus took the lead. In the meantime, Ryan C. Barnett took charge of the ModSecurity rules and produced significant improvements with Core Rule Set v2. In 2010, Trustwave acquired Breach Security and promised to revitalize ModSecurity. The project was then handed to Ryan C. Barnett and Breno Silva.

Something remarkable happened in March 2011: Trustwave announced that it would change the license of ModSecurity from GPLv2 to Apache Software License (ASLv2). This was 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 the Open Web Application Security Project (OWASP). Subsequently, commercial WAF offerings started to incorporate the ModSecurity engine and added the OWASP ModSecurity Core Rules as a default ruleset. With version 2.7.0, ModSecurity was ported to work with Nginx and IIS web servers, but these ports never achieved the stability of the original version. This eventually led to a major rewrite that would be able to support multiple platforms equally well. That will become ModSecurity 3.0, currently in the making.

In 2013, Felipe Costa took over the lead developer position from Breno, and when Ryan left Trustwave in 2015 he handed over the rules to Chaim Sanders, who joined Trustwave in 2014 to support the project with coding and community management.

What Can ModSecurity Do?

ModSecurity is a toolkit for real-time web application monitoring, logging, debugging, and access control. I like to think of it as an enabler. There are no hard rules telling you what to do; instead, it's 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 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. This is a matter not of ideology, but of practicality. I simply don't want my tools to restrict what I can do.

The following is a list of the most important usage scenarios for ModSecurity:

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 can block reliably, if you so wish, because ModSecurity uses full request and response buffering.

Virtual patching
: *Virtual patching* is a concept that addresses vulnerability mitigation in a separate layer, in which 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's particularly useful with HTTP, because traffic generally can 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. Virtual patching is, by far, the activity ModSecurity offers that requires the least investment, is the easiest to perform, and 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 can't get all the data that you need. I have yet to encounter a web server that is able to log full transaction data—but ModSecurity gives you the ability to log everything, 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. As a bonus, this type of detailed logging is also helpful for application troubleshooting—not just security.

Continuous passive security assessment

Security assessment is seen largely 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 in which 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're 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. For example, it's 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 makes unusual demands of us, and when handling such demands, the flexibility of ModSecurity comes in handy when you need it the most. You may have to address a security need, or maybe you have a completely different issue; 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'm often 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's 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.

Guiding Principles

There are three guiding principles on which ModSecurity is based:

Flexibility
: ModSecurity was designed and built with a particular user in mind: a security expert who needs to be able 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 providing 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: granular control down to the individual byte.

Passiveness
: Another key design decision was to make ModSecurity as passive as possible; it will thus never make changes to transaction data unless instructed to do so. The key reason for this was to give users confidence to deploy ModSecurity with entirely passive rulesets that allow them to just observe, safe in knowing that their applications will not be affected. 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.

There are elements in ModSecurity that fall outside the scope of these principles. For example, ModSecurity can change the way Apache identifies itself to the outside world, confine the Apache process within a jail, and even inject security tokens into the traffic. Although these functions are useful, I think that they detract from the main purpose of ModSecurity, which is to be a reliable and predictable tool that enables 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 of 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, ideally from the 2.4.x branch. That said, a version from the 2.2.x branch will also work. ModSecurity has been ported to Nginx and to IIS, which introduces wider platform options. 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 preferred 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 of 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 which you get complete isolation from the systems you are protecting. On the performance front, a standalone ModSecurity installation 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.

Getting Started

In this first practical section of the book, I will give you a whirlwind tour of ModSecurity's internals to help you get started.

Hybrid Nature of ModSecurity

ModSecurity is a hybrid WAF engine that relies on the host web server for some of its work. ModSecurity was originally written for the Apache web server but has since been ported to Nginx and to IIS. Although both ports are actively maintained, they suffer from ModSecurity's heritage and tight integration with the Apache source code. The next major version of ModSecurity is being reimplemented to separate it from Apache, allowing it to support all web servers equally well. Until that happens, the best web server to run ModSecurity is Apache 2.x.

Apache does for ModSecurity what it does for all other modules—it handles the following 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
5. Dechunks request bodies as necessary

There are 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. Dechunks response bodies as necessary

The advantage of a hybrid implementation is that it's 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 normally isn't a problem, because what Apache doesn't see can't harm any module or application. In some rare cases, however, the purpose of NUL-byte evasion is to hide something, 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 in embedded mode; they can't 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 ModSecurity usually sees complete requests before they're passed to the application for processing, and complete responses before they're sent to clients. Buffering is an

important feature, because it's 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's 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 doing: 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're 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 in which your attackers don't have anything to lose but have everything to win. You, on the other hand, don't have anything to win but everything to lose.

What Rules Look Like

Every part of 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's too early to go into how the rules work, I'll include a quick example here just to give you an idea of what they look like. For example:

```
SecRule ARGS "@rx <script>" \
    "id:2000,log,deny,status:404"
```

Even without further assistance, you can probably recognize the part in the rule that specifies what we want to look for in input data (`<script>`). Similarly, you'll easily figure out what will happen if we do find the desired pattern (`log,deny,status:404`). Things will become more clear when you look at the general rule syntax, as follows:

```
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, indicates 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 is used to add metadata to the rules and to specify what ModSecurity should do when a match occurs. The rule from the previous example assigns ID 2000 to uniquely identify the rule and specifies the following actions on a match: log problem, stop transaction processing, and return HTTP response code 404.

I hope you aren't 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 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 perform some work at the beginning (e.g., parse data that has become available), invoke the rules specified to work in that phase, and perhaps perform a task 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 that each phase exists. There is always one task, sometimes several, that can only be performed 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 in this phase is the time to do it. For example, ModSecurity will not parse an XML or JSON 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. Afterward, the web server will either generate the response itself (in embedded mode) or forward the transaction to a backend web server (in reverse proxy mode).

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 and all its data made available for the rules to make their decisions.

Logging (5)
: The logging phase is special. 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 performed or to save information in persistent storage.

Lifecycle Example

To give you a better idea of what happens in every transaction, we'll examine a detailed debug log of one POST transaction. The debug log is an additional logging facility provided by ModSecurity that allows you to observe the execution steps of the module in great detail. I've deliberately chosen a transaction type that uses the request body as its principal method to transmit data, because 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 log line.

> **Note**
>
> Please don't try to understand everything about the logs at this point. The idea is to get a general feel for how ModSecurity works and an introduction to debug logs. Soon after you start to use ModSecurity, you'll discover that debug logs are an indispensable rule-writing and troubleshooting tool.

The transaction I'm 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's 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: Fri, 22 Jul 2016 04:59:13 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`. With this, 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 in the request line and in the request headers. In this example, the query string part contains a single parameter (`a`), so you'll see a message documenting its discovery. ModSecurity will then create a transaction context and invoke the `REQUEST_HEADERS` phase:

```
[4] Initialising transaction (txid V5LjWH8AAQEAAFPTr64AAAAA).
[5] Adding request argument (QUERY_STRING): name "a", value "test"
[4] Transaction context created (dcfg 1154668).
[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's 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 1154668).
[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 mentioned in the logs are parts of ModSecurity that handle request and response bodies:

```
[4] Hook insert_filter: Adding input forwarding filter (r 7f5fc8002970).
[4] Hook insert_filter: Adding output filter (r 7f5fc8002970).
```

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 7f5fc800ae90, r 7f5fc8002970).
[4] Input filter: Forwarded 6 bytes.
```

```
[4] Input filter: Sent EOS.
[4] Input filter: Input forwarding complete.
```

The request is now in the hands of Apache's request handler. If the web server is running in embedded mode, the request handler will generate the response itself. If it's running in reverse proxy mode, the server will forward the transaction to a backend server.

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 7f5fc800aeb8, r 7f5fc8002970).
[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] Content Injection: Not enabled.
[9] Output filter: Bucket type MMAP contains 13 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] Recording persistent data took 0 microseconds.
[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: ------------------------ce3de83f6cf79943
[9] Input filter: Bucket type HEAP contains 140 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
[9] Input filter: Bucket type HEAP contains 116 bytes.
[4] Multipart: Created temporary file 1 (mode 0600): /usr/local/modsecurity/var/tmp/20160723-054018-V5l
[9] Multipart: Added file part 7f67b400fd50 to the list: name "f" file name "eicar.com.txt" (offset 140
[9] Input filter: Bucket type EOS contains 0 bytes.
[4] Request body no files length: 96
[4] Input filter: Completed receiving request body (length 256).
```

In addition to seeing the multipart parser in action, you'll 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'll see the cleanup and the temporary file deleted:

```
[4] Multipart: Cleanup started (remove files 1).
[4] Multipart: Deleted file (part) "/usr/local/modsecurity/var/tmp/20160723-054427-V5LoG38AAQEAAF4SFAoA
```

The temporary file won't 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 "/usr/local/modsecurity/var/tmp/20160723-054018-V5LnIn8AAQEAAFurQfQAA
```

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 6080 bytes.
[9] Input filter: Bucket type HEAP contains 2112 bytes.
[9] Input filter: Bucket type HEAP contains 5888 bytes.
[9] Input filter: Bucket type HEAP contains 2304 bytes.
[9] Input filter: Bucket type HEAP contains 5696 bytes.
[9] Input filter: Bucket type HEAP contains 2496 bytes.
[9] Input filter: Bucket type HEAP contains 5504 bytes.
[9] Input filter: Bucket type HEAP contains 2688 bytes.
[9] Input filter: Bucket type HEAP contains 5312 bytes.
[9] Input filter: Bucket type HEAP contains 2880 bytes.
[9] Input filter: Bucket type HEAP contains 5120 bytes.
[9] Input filter: Bucket type HEAP contains 3072 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: /usr/local/modsecurity/var/tmp/2016072
[4] Input filter: Wrote 128146 bytes from memory to disk.
```

This file is always deleted in the cleanup phase:

```
[4] Multipart: Deleted file (part) "/usr/local/modsecurity/var/tmp/20160723-054813-V5Lo-X8AAQEAAF4SFAsA
```

```
[4] Input filter: Removed temporary file: /usr/local/modsecurity/var/tmp/20160723-054813-V5Lo-X8AAQEAAF
```

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—namely, the rules—and the usage of your server. The following is a detailed list of the various activities that increase resource consumption:

- ModSecurity will add to the parsing already performed by Apache, which results in a slight increase in 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 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 CPU time (for the operator) and RAM (to transform input data before it can be analyzed).
- Some operators used in the rules (e.g., the regular expression operator) are CPU-intensive. Running regular expressions on very large request or response bodies can take a long time—seconds, even.
- 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 one person may not be to someone else. If you don't have enough resources to do everything you want with ModSecurity, you'll need to monitor the operation of your system and remove some functionality to reduce the resource consumption; virtually everything that ModSecurity does is configurable, so you should have no problems doing so.

It's 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 performed 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's Next?

The purpose of this section is to map your future ModSecurity activities and help you determine where to go from here. Where you'll 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're done with those tasks, you need to decide what you want to do with ModSecurity—and 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 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 available information to make decisions to warn or block), write virtual patches, and even to check on the status of ModSecurity itself.

It's true that the attack detection rules are in a class of their own, but that's mostly because in order to write them successfully, you need to know a great deal about application security. For that reason, many ModSecurity users generally focus on using third-party rulesets for 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.

Rulesets

The use of existing rulesets 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 CRS 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'll know when something happens and can visit the audit logs to in-

vestigate. 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'll probably aim to install one of the available remote centralization tools and use its GUI. The available options are listed in the following Resources section.

Resources

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

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.[1] 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.[2]

Issue tracker
: You'll want to visit the ModSecurity issue tracker[3] for one of two reasons: to report a problem with ModSecurity itself (e.g., when you find a bug) or to check the progress on the next (major or minor) version. Before reporting any problems, go through the *support checklist*,[4] which will help you assemble the information required to help resolve your problem. Providing as much information as possible 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 through which you can discuss ModSecurity.[5] Feel free to ask questions, propose improvements, and discuss ideas. You'll hear about new ModSecurity versions first through this list.

---

[1] ModSecurity web site (SpiderLabs, retrieved 29 December 2016)

[2] ModSecurity documentation (SpiderLabs, retrieved 29 December 2016)

[3] ModSecurity issue tracker (GitHub, retrieved 29 December 2016)

[4] ModSecurity Support Checklist (SpiderLabs, retrieved 29 December 2016)

[5] ModSecurity Users' mailing list (SourceForge, retrieved 29 December 2016)

Core Rule Set mailing list

The CRS project[6] is part of OWASP[7] and has a separate mailing list (*owasp-modsecurity-core-rule-set@lists.owasp.org*). Discussions about false positives and the development of new rules also take place in the Core Rules GitHub repository.[8]

Developer Resources

If you're interested in development work, you'll need to access the following resources:

Developers' mailing list

The developers' mailing list is a resource for discussing ModSecurity software development.[9] If you do decide to start playing with the source code, use this list to seek advice and to discuss your work. There is also a ModSecurity developers' guide available with guidelines and examples.[10]

Source code access

The source code of ModSecurity is hosted in a GitHub repository, which allows you to access it directly or through a web-based UI.[11]

AuditConsole

Using ModSecurity entirely from the command line is a lot of fun, but reviewing audit and debug logs is difficult without special scripts or higher-level tools. Your best choice for a log centralization and GUI tool is AuditConsole, built by Christian Bockermann.[12]

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 triggers, which are executed in the console

Summary

This chapter provided a ModSecurity orientation. I introduced ModSecurity at a high level, discussed what it is and what it isn't, and what it can and can't do. I also gave you a taste of

[6] Core Rules Project (OWASP, retrieved 29 December 2016)

[7] OWASP (OWASP, retrieved 29 December 2016)

[8] CRS GitHub repository (GitHub, retrieved 29 December 2016)

[9] ModSecurity developers' mailing list (SourceForge, retrieved 29 December 2016)

[10] ModSecurity developers' guide (SpiderLabs, retrieved 29 December 2016)

[11] ModSecurity source code (GitHub, retrieved 29 December 2016)

[12] AuditConsole (Christian Bockermann, retrieved 15 January 2017)

what ModSecurity is like and described its 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 option comes with advantages and disadvantages, as listed in Table 2.1.

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	• Semiautomated installation	• May not be the latest version • Manual download and updates • Must determine if 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 successive maintenance of a custom-installed program?
- Will you need to make changes to ModSecurity or write your own extensions?

Casual users should generally try to use binary packages when they're available (and they are available in most distributions).

Installation from Source

When we build dedicated reverse proxy installations, we tend to build everything from source, because that allows us access to the latest Apache and ModSecurity versions and makes it easier to tweak elements (by changing the source code of either Apache or ModSecurity) when we want to.

Downloading Releases

To download ModSecurity, go to its web site[1] or the GitHub project page.[2] You will need both the main distribution of the source code and its cryptographic signature:

```
$ wget https://www.modsecurity.org/tarball/2.9.1/modsecurity-2.9.1.tar.gz
$ wget https://www.modsecurity.org/tarball/2.9.1/modsecurity-2.9.1.tar.gz.asc
```

Verify the signature before doing anything else, to ensure the package you've just downloaded doesn't contain a Trojan horse planted by a third party and that it hasn't been corrupted during transport:

```
$ gpg --verify modsecurity-2.9.1.tar.gz.asc
gpg: Signature made Wed 09 Mar 2016 19:48:15 CET using DSA key ID E8B11277
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 --keyserver pgp.mit.edu --recv-keys E8B11277
gpg: requesting key E8B11277 from hkp server pgp.mit.edu
gpg: key E8B11277: public key "Felipe Zimmerle da Nobrega Costa ↩
<felipe@zimmerle.org>" imported
gpg: 3 marginal(s) needed, 1 complete(s) needed, classic trust model
gpg: depth: 0  valid:   3  signed:   5  trust: 0-, 0q, 0n, 0m, 0f, 3u
gpg: depth: 1  valid:   5  signed:   5  trust: 2-, 0q, 0n, 2m, 1f, 0u
gpg: depth: 2  valid:   4  signed:   0  trust: 1-, 0q, 0n, 0m, 3f, 0u
gpg: next trustdb check due at 2018-09-26
gpg: Total number processed: 1
gpg:               imported: 1
```

Now you can try again:

```
$ gpg --verify modsecurity-2.9.1.tar.gz.asc
gpg: Signature made Wed 09 Mar 2016 19:48:15 CET using DSA key ID E8B11277
gpg: Good signature from "Felipe Zimmerle da Nobrega Costa ↩
<felipe@zimmerle.org>"
```

[1] ModSecurity web site (SpiderLabs, retrieved 29 December 2016)

[2] ModSecurity GitHub page (GitHub, retrieved 29 December 2016)

```
gpg:                 aka "Felipe Zimmerle"
gpg:                 aka "Felipe Costa <fcosta@trustwave.com>"
gpg:                 aka "Felipe Zimmerle (gmail) <zimmerle@gmail.com>"
gpg:                 aka "[jpeg image of size 7280]"
gpg:                 aka "[jpeg image of size 14514]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 190E FACC A1E9 FA46 6A8E  CD9C E6DF B08C E8B1 1277
```

The warning in the previous snippet might look serious, but it generally isn't a problem; it 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 Felipe'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 Felipe 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 GitHub repository (the source code control system used by the ModSecurity project) is the way to go. When you do so, 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 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.

Before you can install a development version of ModSecurity, you need to know where to find it. The repository, which is hosted with GitHub, can be viewed with a browser.[3]

The default view on GitHub is the `master` source code tree, which shows the most recent development version with the latest accepted changes. Proposed changes are accessible via pull requests or via their own separate branch. These active branches may sometimes contain a feature or a fix that has not been accepted into the master source code. If you want to download a release candidate or a tested release, you can access these archives via a separate submenu.

Once you've determined the location of the version of ModSecurity you want to use, you can get it using the `clone` command of Git, like this:

```
$ git clone https://github.com/SpiderLabs/ModSecurity.git modsecurity-master
```

What you'll get in the `modsecurity-master` folder is almost the same as what you get when you download a release. Some files need to be generated via a special command first, though. Furthermore, the documentation might not be in sync. The master documentation is kept in a wiki, with copies of the wiki included with releases.

[3] ModSecurity source code repository (GitHub, retrieved 29 Dec 2016)

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're using for instructions. If you'll be adding ModSecurity to an operating system–provided Apache, you're likely to need to install a specific Apache development package too. For example, on Debian and Ubuntu you need to use `apache2-dev`.

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

Table 2.2. ModSecurity dependencies

Dependency	In Apache?	Purpose
Apache Portable Runtime (APR)[a]	Yes	Various
APR-Util[b]	Yes	Various
`mod_unique_id`	Yes, but may not be installed by default	Generate unique transaction ID
libcurl[c]	No	Remote logging (`mlogc`)
libxml2[d]	No	XML processing
Lua[e]	No	Writing complex rules in Lua (optional)
Perl Compatible Regular Expressions (PCRE)[f]	Yes, but cannot be used by ModSecurity	Regular expression matching
ssdeep[g]	No	Perform fuzzy hash matching
YAJL[h]	No	JSON processing and JSON format logging

[a] Apache Portable Runtime (Apache Portable Runtime Project, retrieved 29 December 2016)

[b] APR-Util (Apache Portable Runtime Project, retrieved 29 December 2016)

[c] libcurl (libcurl, retrieved 29 December 2016)

[d] libxml2 (xmlsoft.org, retrieved 29 December 2016)

[e] Lua 5.2 (Lua.org, retrieved 29 December 2016)

[f] Perl Compatible Regular Expressions (PCRE, retrieved 29 December 2016)

[g] ssdeep (SourceForge, retrieved 29 December 2016)

[h] YAJL (GitHub, retrieved 29 December 2016)

If you already have Apache installed, you'll only ever need to deal with libcurl, libxml2, Lua, ssdeep, and YAJL. With Apache compiled from source, you'll also need the PCRE library. Apache no longer comes bundled with it. To work around this issue, install PCRE separately and then tell Apache to use the external copy; I explain how to do so 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 distributions from the Debian family, the following command installs the missing packages:

```
# apt-get install libcurl3-dev liblua5.3-dev libxml2-dev libfuzzy-dev libyajl-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 GnuTLS. You are advised to use OpenSSL because there have been complaints about remote logging problems when GnuTLS was used. APR-Util is usually compiled without support for cryptographic operations. If you want to use the directive `SecRemoteRule` with the parameter `crypto`, you'll need to compile APR-Util yourself.

The process should be straightforward from here on. If you cloned the GitHub repository and did not download a release, then you need to generate the configuration script, which is used to prepare the compilation process:

```
$ ./autogen.sh
```

If you downloaded a release, then you can skip this step and execute the following commands directly in succession:

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

This set of commands assumes that you don't need any compile-time options. If you do, see the following subsection.

> **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're 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 doesn't activate it; you must do that manually. (While you're doing so, 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 runtime 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 on its own. It may or may not do so, but chances are good that you'll occasionally need to do something different; this is where the compile-time options listed in Table 2.3 come in handy.

Table 2.3. Main compile-time options

Option	Description
`--disable-request-early`	Shift the first processing phase of ModSecurity to a later position in the lifecycle of an Apache request. The default is to run the phase early.
`--with-apr`	Specify the location of the Apache Portable Runtime library.
`--with-apu`	Specify the 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.
`--with-pcre`	Specify the location of PCRE.
`--with-ssdeep`	Specify the location of ssdeep.
`--with-yajl`	Specify the location of YAJL.

There are a few additional options dealing with the audit log format of ModSecurity. They are rarely used in practice, but take a look at the configure script to get an overview.

Custom-Compiled Apache Installations

Using ModSecurity with a custom-compiled version of Apache is straightforward. With Apache 2.2, there used to be issues with PCRE and the `mod_unique_id` module not being enabled by default, but these were solved with Apache 2.4.

To configure ModSecurity, use the `--with-apxs` compile-time option to specify the location of your Apache installation. In the following example, I'm assuming Apache is installed in `/usr/local/apache`:

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

From here, 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 /usr/local/apache/bin/httpd  | grep pcre
        libpcre.so.3 => /lib64/libpcre.so.3 (0x00007ff2a11fd000)
```

You should get the same result when you compile ModSecurity:

```
$ ldd /usr/local/apache/modules/mod_security2.so | grep pcre
        libpcre.so.3 => /lib64/libpcre.so.3 (0x00007f85995c5000)
```

> **Tip**
>
> Mac OS X does not have `ldd`, but you can obtain the equivalent functionality by running `otool` with option `-L`. If you really get stuck, consider using `install_name_tool` to change library dependencies after ModSecurity is compiled.

It is quite possible to have a configuration in which Apache uses its bundled PCRE and ModSecurity uses another PCRE version available on the system.

ModSecurity reports the detected library version numbers at startup (in the error log) 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 various library collisions, which can happen in odd situations.

```
ModSecurity for Apache/2.9.1 (http://www.modsecurity.org/) configured.
ModSecurity: APR compiled version="1.5.2"; loaded version="1.5.2"
ModSecurity: PCRE compiled version="8.39 "; loaded version="8.39 2016-06-14"
ModSecurity: LUA compiled version="Lua 5.2"
ModSecurity: YAJL compiled version="2.0.4"
ModSecurity: LIBXML compiled version="2.9.1"
```

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 limited to an older version. Further, packagers often do not include `mlogc`, which is helpful for remote log centralization. In general, if you're okay with the way the module was compiled, then you'll be fine with binary packages.

Fedora Core, CentOS, and Red Hat Enterprise Linux

If you're a Fedora user, you can install ModSecurity directly from the official distribution, using `yum`:

```
# yum install mod_security
```

On CentOS and Red Hat Enterprise Linux, you have to use the packages from Extra Packages for Enterprise Linux (EPEL), a volunteer effort that's part of the Fedora community.[4] The installation process is the same as for Fedora.

Debian and Ubuntu

Debian was the first distribution to include ModSecurity. Alberto Gonzalez Iniesta has been a long-time supporter of ModSecurity on Debian, supporting ModSecurity in his own (unofficial) repository and later becoming the official packager.

If you are running a version of the Debian family, the installation is easy:

```
# apt-get install libapache2-mod-security2
```

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 of naming configuration files 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 the 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.[5] 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'll have little trouble if you want to run the latest version. The download includes ModSecurity and `mlogc`.

> **Note**
>
> Although it might be possible to run Steffen's ModSecurity binaries with a version of Apache produced elsewhere, you really should use only the packages from a single location that are intended to be used together. Otherwise, you may encounter unusual behavior and web server crashes.

[4] Extra Packages for Enterprise Linux (Fedora Project, retrieved 29 December 2016)

[5] Apache Lounge web site (Apache Lounge, retrieved 29 December 2016)

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's included with so many operating systems and distributions. Although 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're not used to it; it's not everyone's cup of tea. There's 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 installed 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 most of 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 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'm 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 (with the exception of the logging directives, which are listed in several tables in Chapter 4, *Logging*).

Table 3.1. Main configuration directives

Directive	Description
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
SecUploadDir	Sets the folder in which intercepted files will be stored
SecUploadFileLimit	Set the maximum number of file uploads processed in a multipart POST
SecUploadKeepFiles	Controls whether the uploaded files will be kept after the transaction is processed

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's difficult to give specific advice. Similarly, different choices may be appropriate in different circumstances. For example, if you're 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 always prefer to 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 (`/usr/local/modsecurity`) with multiple subfolders underneath. The subfolders that are written to at runtime are all grouped (in `/usr/local/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
`/usr/local/modsecurity/bin`

Configuration files/rules
`/usr/local/modsecurity/etc`

Audit logs
`/usr/local/modsecurity/var/audit`

Persistent data
`/usr/local/modsecurity/var/data`

Logs
`/usr/local/modsecurity/var/log`

Temporary files
`/usr/local/modsecurity/var/tmp`

File uploads
`/usr/local/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 also means `root` must be the principal owner of the installation. Because it's not good practice to stay as `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.

Table 3.2. Folder permissions

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

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

1. As already discussed, `root` that owns everything by default, and we assign ownership to `apache` only when necessary.
2. In two cases (`/usr/local/modsecurity` and `/usr/local/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 /usr/local/modsecurity/bin folder, where you might store some binaries Apache will need to execute at runtime.

3. One folder, /usr/local/modsecurity/var/log, stands out; it's the only folder underneath /usr/local/modsecurity/var to which apache is not allowed to write. That folder contains log files that are opened by Apache early on while it's 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. Anything else 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's 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 /usr/local/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'll 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 via ClamAV). To allow the third user account to access the files created by ModSecurity, 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. The layout depends on what you want to do with ModSecurity. If you plan to run the OWASP ModSecurity Core Rule Set, for example, you'll follow their setup proposal to a certain extent. Other rule layout conventions have more to do with taste than anything else, but in this section I'll describe an approach that's 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.

Table 3.3. Configuration files

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

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

```
Include /usr/local/modsecurity/etc/main.conf
Include /usr/local/modsecurity/etc/rules-first.conf
Include /usr/local/modsecurity/etc/rules.conf
Include /usr/local/modsecurity/etc/rules-last.conf
```

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 Lua
LoadFile /usr/lib/x86_64-linux-gnu/liblua5.3.so
# Finally, load ModSecurity
LoadModule security2_module modules/mod_security2.so
```

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

```
<IfModule mod_security2.c>
    Include /usr/local/modsecurity/etc/modsecurity.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; it allows you to deactivate a module simply by commenting out the appropriate `LoadModule` line.

> **Note**
>
> Prior to Apache 2.4, it was necessary to load the `libxml2.so` file similar to `liblua5.2.so`. However, this is no longer the case, because the XML library is now linked into the ModSecurity module directly.

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 can be sure nothing will be blocked:

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

You'll 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'll 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 can 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.

Request Body Handling

Requests consist of two parts: the headers part, which is always present, and the body, which is optional and depends on the HTTP method employed. 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 not only will have access to the content transmitted in request bodies but also will 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; 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 occurs. 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
```

File uploads generally don't use RAM (and thus don't create an opportunity for a memory-based denial of service attack), which means that it's safe to allow large requests as defined by `SecRequestBodyLimit`. With all the other requests, the RAM usage has to be considered, and a lower limit is imperative. `SecRequestBodyNoFilesLimit` is applied in such cases.

> **Warning**
>
> When the `SecStreamInBodyInspection` directive 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 for more information.

The third directive that addresses 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 up the bulk of the traffic on most web sites, most of which is just static files that don't have any security relevance in most cases. The response MIME type is used to distinguish interesting responses from those that are not. The `SecResponseBodyMimeType` directive lists the response MIME types you're interested in:

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

> **Note**
>
> To instruct ModSecurity to inspect response bodies for which the MIME type is unknown (meaning that it was not specified in the response headers), use the special string (`null`) as a parameter for `SecResponseBodyMimeType`.

You can control the size of a response body buffer via 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 for which 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 first glance, 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's long enough to bypass observation by ModSecurity—and this is true. However, if you have an

attacker with full control of output, it's 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 cases in which 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.

Dealing with Response Compression

When deploying ModSecurity in reverse proxy mode with backend servers that support compression, make sure to set the `SecDisableBackendCompression` directive to `On`. Doing so will hide the fact that the clients support compression from your backend servers, giving ModSecurity access to uncompressed data. If you don't disable backend compression, ModSecurity will see only the compressed response bodies (as served by the backend web servers). To continue to use frontend compression, configure `mod_deflate` in the proxy itself. The `SecDisableBackendCompression` directive will not interfere with its operation.

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
# (e.g., when it needs to handle a multipart request
# body that's larger than the configured limit). If you don't
# specify a location here, your system's default will be used.
# It's recommended that you specify a location that's private.
SecTmpDir /usr/local/modsecurity/var/tmp/

# The location where ModSecurity will keep its data. This,
# too, needs to be a path that other users can't access.
# IMPORTANT: The path defined by SecDataDir must reside on
# on the same partition as the path defined by SecTmpDir.
SecDataDir /usr/local/modsecurity/var/data/
```

File Uploads

Next, we'll configure the handling of file uploads. We'll configure the folder where ModSecurity will store intercepted files, but keep this functionality disabled for now. File upload inter-

ception slows down ModSecurity and can potentially consume a lot of disk space, so you'll want to enable this functionality where you really need it.

```
# The location where ModSecurity will store intercepted
# uploaded files. This location must be private to ModSecurity.
SecUploadDir /usr/local/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
# don't allow any other user to access them. You may need to
# relax that if you want to interface ModSecurity with an
# external program (e.g., an anti-virus program).
SecUploadFileMode 0600
```

You should set 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
```

There isn't a limit by default, so setting one in the configuration is very important. The issue here first is that it's easy for an attacker to include many embedded files (hundreds or even thousands) in a single `multipart/form-data` request, but also 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 performance. The debug log will duplicate what you'll also see in Apache's error log up to level 3. If the error log is growing fast and has to be rotated quickly, it can be useful to keep the ModSecurity-related messages longer in the debug log. However, if there are a lot of ModSecurity alerts, redundancy will be an issue, and you'll need to make sure you also rotate the debug log regularly. It's perfectly okay to run with a debug log level of 0 and to rely on the Apache error log. Any value above 3 is not recommended in production.

```
# Debug log
SecDebugLog /usr/local/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're 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's really necessary.
SecAuditEngine RelevantOnly
```

In addition, we'll 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
```

Alternatively, you can also log client errors in the range of 400–499. This can be useful because affected users often will contact you with support questions. You probably don't want to log status code 404 (Not Found) in this case, so the complete regular expression to keep an audit log of all erroneous requests—with the exception of 404—is as follows:

```
# Also log requests that cause an error.
SecAuditLogRelevantStatus "^(?:5|4(?!04))"
```

The audit log separates its records into multiple parts. Each part is assigned a single letter. You enable the logging of the individual parts by listing them as parameters of the `SecAuditLogParts` directive. 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 installations that will log a lot and it prevents remote logging, but it's good enough to start with:

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

As the final step, we'll 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 /usr/local/modsecurity/var/audit/
```

Default Rule Match Policy

Now that we're nearing the end of the configuration, you need to decide what you want to happen when a rule matches. We recommend 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's possible to write rules that ignore the default policies. If you're using third-party rulesets and 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 it to you to deal with them. In our default configuration, we'll have a couple of rules to deal with the situations that ModSecurity can't deal with on its own: processing errors.

> **Note**
>
> I'm including these rules here because they should be an integral part of every configuration, but you shouldn't worry if you don't understand exactly what it is that they do. The mechanics of the rules will be explained in detail later in the book.

There are currently three types of processing errors:

1. Request and response buffering limits encountered
2. Parsing errors
3. 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 can detect the processing errors.

Currently, the only parsing errors that can happen are request body processor errors. We'll use two rules to handle such errors. 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" \
    "id:2000,phase:2,block,t:none,log,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" \
"id:2001,phase:2,block,t:none,log,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 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 ModSecurity users' mailing list, noting that you've encountered an interesting attacker or a ModSecurity bug.

PCRE limits are set to protect the server from denial of service attacks via excessive resource consumption in regular expression calculations. The default limits are very low. Therefore, users can control the setting of the limits via `SecPcreMatchLimit` and `SecPcreMatchLimitRecursion`. The debug log will identify rules that have exceeded the limits—for example:

```
[3] Rule 292d670 [id "941140"][file "/usr/local/modsecurity/etc/core-rules/REQUEST-941-APPLICATION-ATTA
```

For now, leave the PCRE limits defaults as they are, but add a rule to warn us when they're exceeded:

```
SecRule TX:MSC_PCRE_LIMITS_EXCEEDED "@eq 1" \
    "id:9000,phase:5,pass,t:none,log,msg:'PCRE limits exceeded'"
```

I've used phase 5 for the rule, but if you're 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, we recommend undertaking 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 string `MY_UNIQUE_TEST_STRING`, responding with a 503 (Service Unavailable) on a match:

```
SecRule ARGS "@contains MY_UNIQUE_TEST_STRING" \
    "id:2000,phase:2,deny,status:503,log"
```

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're 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's 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 never be 100% certain that a third-party module was implemented correctly. For exam-

ple, it's possible to write a module that will essentially hijack a request early on and bypass all other modules, including ModSecurity. We're 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're correct.

You're done!

Summary

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

Quite a few more optional configuration directives exist. Most of them are highly advanced or only applicable in rare and special situations. They're covered in Chapter 15, *Directives*.

We didn't pay much attention to logging in this chapter, opting to configure both the debug log and the audit log conservatively. However, 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 chapter covers the logging capabilities of ModSecurity in detail. Logging is a big part of what ModSecurity does, so it's not surprising that there are extensive logging 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're 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.

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. Messages with levels 1–3 are copied to Apache's error log. The higher-level messages are there mostly for troubleshooting and debugging.

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 having the messages only in the error log). 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 transaction consumes a lot of resources.

This is what a single debug log line looks like:

```
[24/Jul/2016:17:38:25 +0200] [192.168.3.111/sid#15e21f8][rid#7f683c002970]↵
```

```
[/index.html][4] Initialising transaction (txid V5Tg8X8AAQEAABXZdiwAAAAA).
```

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.

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., nonblocking rule matches)
3	Notices (e.g., nonfatal processing errors)
4	Handling of transactions and performance
5	Detailed syntax of the rules
6–8	Not used
9	Detailed information about transactions (e.g., variable expansion and setting of variables)

Debugging in Production

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

> **Note**
>
> The audit log can record all the rules that matched during an HTTP transaction. This helpful feature 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 want to debug. You can do this by overriding the default configuration, using the `<Location>` context directive, as shown ahead. While you're doing that, it may be a good idea to specify a different debug log file altogether. That way, you'll keep the main debug log file free of your tests.

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

This approach, although handy, still doesn't 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 a specific client sends. 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 by 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 "@ipMatch 192.168.1.1" \
    id:1000,phase:1,pass,nolog,ctl:debugLogLevel=9
```

Using your IP address won't work in cases in which you're hidden by a network address translation 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's done depends on the browser you're using.)

```
SecRule REQUEST_HEADERS:User-Agent "@contains YOUR_UNIQUE_ID" \
    id:1000,phase:1,pass,nolog,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 to dynamically enable 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 a request-by-request basis, using one of the tools that support request interception and modification. You can do so via a browser extension or can use an interception proxy. 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. While you're at it, it's 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 of lines.

Audit Log

It's a little-known fact that the earliest versions of ModSecurity essentially had one feature: the ability to record complete HTTP transaction data. Being able to see exactly what's exchanged between browsers and servers is very important for developers, yet few web servers make it possible. The audit log, which does just that, was one of the first features implemented.

ModSecurity is currently able to log most, but not all, transactions. Transactions involving errors (e.g., 400 and 404 transactions) and some third-party Apache modules like `mod_auth_cas`

use a different execution path, which ModSecurity doesn't support. This means that they shortcut the ModSecurity rule phases 1 to 4, effectively preventing the module from extracting the necessary data out of the request. Audit log directives are shown in Table 4.3.

Table 4.3. Audit log directives

Directive	Description
`SecAuditEngine`	Controls the audit log engine; possible values are `On`, `Off`, or `RelevantOnly`
`SecAuditLog`	Path to an audit log file
`SecAuditLog2`	Path to another audit log file (copy)
`SecAuditLogDirMode`	Permissions (mode) of the folders created for the concurrent log type
`SecAuditLogFileMode`	Permissions (mode) of the files created for the concurrent log type
`SecAuditLogFormat`	Format of the audit log (native or JSON)
`SecAuditLogParts`	Specifies which part of a transaction will be logged
`SecAuditLogRelevantStatus`	Specifies which response statuses will be considered relevant
`SecAuditLogStorageDir`	Path where 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, brief `User-Agent`, without a body, and no logging of the response body) consumes around 1.5 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. There is a native format for the individual entries or an alternative JSON format. Looking at a single audit log entry in the default native format, 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 parts A and Z are mandatory; the use of the other parts is controlled with the SecAuditLogParts directive. Table 4.4 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	Reduced multipart request body, which excludes files (alternative to part C)
J	Information on uploaded files (multipart requests)
K	Contains a list of all rules that matched for the transaction
Z	Final boundary (mandatory)

Native Format 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:

```
--be58b513-A--
[27/Jul/2016:05:46:16 +0200] V5guiH8AAQEAADTeJ2wAAAAK 192.168.3.1 50084 192.168.3.111 80
```

Part B contains the request headers and nothing else:

```
POST /index.html?a=test HTTP/1.1
Host: example.com
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
Referer: http://example.com/index.html
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 6
```

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

```
--be58b513-C--
b=test
```

Part F contains the response headers:

```
--be58b513-F--
HTTP/1.1 200 OK
Last-Modified: Sun, 24 Jul 2016 15:24:49 GMT
ETag: "2d-5386344b7871a"
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:

```
--be58b513-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:

```
--be58b513-H--
Stopwatch: 1470025005945403 1715 (- - -)
Stopwatch2: 1470025005945403 1715; combined=26, p1=0, p2=0, p3=0, p4=0, p5=26, ↩
sr=0, sw=0, l=0, gc=0
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.9.1 (http://www.modsecurity.org/).
Server: Apache
Engine-Mode: "ENABLED"
```

Part K contains a list of rules that matched in a transaction. It isn't unusual for this part to be empty, but if you have a complex ruleset, it may show quite a few rules. Audit logs that record transactions for 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:

```
--be58b513-K--
SecAction "phase:2,auditlog,log,pass,msg:'Matching test'"
```

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

```
--be58b513-Z--
```

JSON Format Audit Log

The JSON audit log format was introduced with ModSecurity 2.9.1. It's enabled via the `SecAuditLogFormat` directive. It's conceptually similar to the native format, but organizes information in a different way. The native audit log parts can be roughly mapped to the JSON format blocks in the following way:

- `Transaction`: A
- `Request`: B and C
- `Uploads`: J (optional)
- `Response`: E and F
- `Matched_rules`: K
- `Audit_data`: H

More information about this additional audit log format is available in the *Data Formats* chapter.

Concurrent Audit Log

Initially, ModSecurity supported only the serial audit logging type. 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 doesn't 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're handled, 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 be created not directly in the folder specified by `SecAuditLogStorageDir` but in an elaborate structure of subfolders, the names of which will be constructed from the current date and time:

```
20160727
20160727/20160727-0546
20160727/20160727-0546/20160727-054616-V5guiH8AAQEAADTeJ2wAAAAK
```

```
20160727/20160727-0546/20160727-054616-V5guBH8AAQEAADTeJ2cAAAAA
```

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 type 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 - - [27/Jul/2016:05:46:16 +0200] "GET / HTTP/1.1" 200 218 "-" "-" V5guiH8AAQE
```

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 twofold:

- The first part, which duplicates some of the information available in audit logs, serves as a record of everything that you've recorded so that you can easily search through it.
- The second part tells you where an audit log entry is stored (e.g., `/20160727/20160727-0546/20160727-054616-V5guiH8AAQEAADTeJ2wAAAAK`), where it begins within that file (always zero, because this feature is not used), how long it is, and its MD5 hash (useful to verify integrity).

When real-time audit log centralization is used, this information isn't written to a file. Instead, it's written to a pipe, which means that it's sent directly to another process, which deals with the information immediately. You'll 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 effective protocol based on HTTP. You'll find it documented in Chapter 20, Data Formats.

If you've followed my installation instructions, you'll have `mlogc` compiled and sitting in your `bin/` folder. To proceed, you'll 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's 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" earlier in this chapter.
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 (the `mlogc-transaction.log` file 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 (to the `mlogc-queue.log` file 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. We usually place it in `/usr/local/modsecurity/etc/mlogc.conf`. First, we need to tell `mlogc` where its "home" is,

which is where it will create its log files. The `mlogc` log files are very important, because—as it's 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  /usr/local/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 `mlogc` 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 /usr/local/modsecurity/var/audit
```

Next, we need to tell `mlogc` where to submit 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 usually 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

# If network communication fails, suspend log
# submission to give the server time to recover.
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'll have to remove them yourself, either manually or with a script.

Activating Remote Logging

You'll 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 "|/usr/local/modsecurity/bin/mlogc ↵
/usr/local/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.

Please note that you still need to configure `SecAuditLogStorageDir` because ModSecurity will refuse to work properly without it. Your complete configuration should look like this now:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^5
SecAuditLogParts ABDEFHIJKZ
SecAuditLogType Concurrent
SecAuditLog "|/usr/local/modsecurity/bin/mlogc ↵
```

```
/usr/local/modsecurity/etc/mlogc.conf"
SecAuditLogStorageDir /usr/local/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  /usr/local/modsecurity/bin/mlogc ↩
/usr/local/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:

```
-rw-r-----  1 root     root  83K Jul 29 15:12 audit.log
-rw-r-----  1 root     root  68M Jul 29 15:12 debug.log
-rw-r--r--  1 root     root  769 Jul 30 05:31 mlogc-error.log
-rw-r--r--  1 root     root    0 Jul 30 05:31 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] [5877/0] Configuring ModSecurity Audit Log Collector 2.9.1.
[3] [5877/0] Delaying execution for 5000ms.
[3] [5897/0] Configuring ModSecurity Audit Log Collector 2.9.1.
[3] [5897/0] Delaying execution for 5000ms.
[3] [5877/0] Queue file not found. New one will be created.
[3] [5877/0] Caught SIGTERM, shutting down.
[3] [5877/0] ModSecurity Audit Log Collector 2.9.1 terminating normally.
[3] [5897/0] Queue file not found. New one will be created.
```

It's 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 5877 and PID 5897). 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, 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—that is, it will report the problem and shut down. (Don't be surprised if Apache continues to attempt to start it. That's what Apache does with piped logging programs.)

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

```
dev:/usr/local/modsecurity/etc# apache2ctl start
[1] [15997/0] Failed to open the error log (null): Bad address
[3] [15997/0] ModSecurity Audit Log Collector 2.9.1 terminating with error 1
```

If `mlogc` manages to open its error log, it will do what's expected and write all error messages there. For example:

```
[1] [16402/0] QueuePath not defined in the configuration file.
[3] [16402/0] ModSecurity Audit Log Collector 2.9.1 terminating with error 1
```

At this point, it's 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 the default logging configuration (level 3), a single audit log entry handled by `mlogc` will produce one line in the log file:

```
[3] [1748/7f65840009c0] Entry completed (0.109 seconds, 1415 bytes): ↩
V5wORn8AAQEAAAbptJUAAABC
```

That's basically all you need to know—that an entry has been safely transmitted to the intended destination. You'll get more information when something goes wrong, of course. For example, you'll see the following message whenever your logging server goes down:

```
[2] [23743/7ff2840009c0] Flagging server as errored after failure to submit entry V5wthX8AAQEAAE1D3OYAA
to loghost port 8888: Connection refused
```

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 see the following information in the log:

```
[3] [1748/7f65840009c0] Clearing the server error flag after successful entry ↩
submission: V5w1WX8AAQEAAAerQowAAADA
[3] [1748/7f65840009c0] Entry completed (0.887 seconds, 1415 bytes): ↩
V5w1WX8AAQEAAAerQowAAADA
```

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. See Table 4.5 for the description of the most common problems.

Table 4.5. Common remote logging problems

Error message	Description
Failed to connect	The server could not be reached. This 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 often, this message means that you configured `mlogc` to submit over plaintext, 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, but such errors are treated as transient.

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, but you've already seen them all in the section called "File Uploads" in Chapter 3.

Table 4.6. File upload directives

Directive	Description
`SecUploadDir`	Specifies the location where intercepted files will be stored
`SecUploadFileLimit`	Specifies the maximum number of file uploads processed
`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 `/usr/local/modsecurity/var/upload`, you'll find files with names similar to these:

```
20160728-102354-V5nBGn8AAQEAACo1bNgAAAAA-file-b9irlZ
20160728-102354-V5nBGn8AAQEAACo1bNkAAAAC-file-TeZQZF
```

You can probably tell that the first two parts of a filename are based on the time of upload, then the unique transaction ID follows, then the `-file-` part that is always the same, and finally a random string of characters at the end. ModSecurity uses this algorithm to generate filenames 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 filename prevents a potential attacker from placing a file with a known name on a server.

When you store a file like this, it's just a file and 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.

There are four separate variables involved with file uploads, and you need to pick the right one for the inspection:

- `FILES`: Collection of filenames as provided by the client, sometimes including the path to the file on the client's filesystem.
- `FILES_NAMES`: Collection of names assigned to the files in the upload request—that is, the names of the form fields describing the files.
- `FILES_TMP_CONTENT`: Collection containing the content of the files.
- `FILES_TMPNAMES`: Collection of temporary filenames and paths on the ModSecurity server.

The best way to inspect a file is to pass the temporary filename to an inspection script, so we settle on the `FILES_TMPNAMES` variable. Once this is clear, a file inspection rule is rather simple:

```
SecRule FILES_TMPNAMES "@inspectFile /usr/local/modsecurity/bin/file-inspect.pl" \
    id:2000,phase:2,block,t:none,log
```

This example rule will invoke the `/usr/local/modsecurity/bin/file-inspect.pl` script 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's 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
```

The alert message in the audit log will only display the verdict, but the debug log will display the full error message. The following debug log lines are produced by the inspection file:

```
[4] Recipe: Invoking rule 2735e58; [file "/usr/local/apache/conf/httpd.conf_pod_2016-07-28_10:10"] [lir
[5] Rule 2735e58: SecRule "FILES_TMPNAMES" "@inspectFile /usr/local/apache/bin/inspect-file.pl" "phase:
[4] Transformation completed in 1 usec.
[4] Executing operator "inspectFile" with param "/usr/local/modsecurity/bin/file-inspect.pl" against FI
[9] Target value: "/tmp//20160728-103559-V5nD738AAQEAAD5bgsAAAAAA-file-yfp59Q"
[4] Executing /usr/local/apache/bin/file-inspect.pl to inspect /usr/local/modsecurity/var/tmp/↵
20160728-103559-V5nD738AAQEAAD5bgsAAAAAA-file-yfp59Q.
[9] Exec: /usr/local/apache/bin/file-inspect.pl
[4] Exec: First line from script output: "1 OK"
[4] Operator completed in 2722317 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: /usr/local/modsecurity/bin/file_inspect.pl
[1] Exec: Execution failed while reading output: /usr/local/modsecurity/bin/file_inspect.pl (End of fil
```

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

Integrating with ClamAV

ClamAV 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:

```
#!/usr/bin/perl

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

if (@ARGV != 1) {
```

```
    print "Usage: modsec-clamscan.pl FILENAME\n";
    exit;
}

my ($FILE) = @ARGV;

$cmd = "$CLAMSCAN --stdout $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 EICAR's web site.[1] 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. The following example shows a successful detection of a "virus":

```
[9] Exec: /usr/local/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamscan: Eicar-Test-Signature"
[4] Operator completed in 2628132 usec.
[2] Warning. File
"/usr/local/modsecurity/var/tmp/20160728-110518-V5nKzn8AAQEAAD5bgsEAAAAC-file-szWj6L" rejected by the a
```

If you look carefully at the example output, you'll see that the inspection took more than two seconds. This isn't unusual (even for my slow virtual server), because we're creating a

[1] EICAR antimalware test file (Retrieved 15 January 2017)

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" in Chapter 3, 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 `apache` group (don't forget to restart the ClamAV daemon to pick up the new group).
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: /usr/local/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamdscan: Eicar-Test-Signature"
[4] Operator completed in 20581 usec.
[2] Warning. File "/usr/local/modsecurity/var/tmp/20160728-111500-V5nNFH8AAQEAAGO1b38AAAAA-file-vSR2dT"
```

Advanced Logging Configuration

By now, you've 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 `auditlog` and `noauditlog` actions don't control logging directly. They should be viewed as mere suggestions; it's 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 on to the next rule.

Dynamic logging configuration
: Rules can make logging decisions that affect entire requests (through the `ctl` action), but that functionality shouldn't be used lightly. Most rules should be concerned only with event generation. 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

The `SecAuditLogParts` directive allows you to configure exactly what parts (how much information) you want logged for every transaction, but one setting won't be adequate in all cases. For example, most configurations won't be logging response bodies, but that information is often required to determine whether certain types of attacks (e.g., XSS) 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 "@rx <script>" \
    id:2000,phase:2,block,log,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 change what you're logging. I know I wouldn't! Luckily, it's a problem that can be resolved with the addition of a phase 5 rule that resets the logged audit log parts:

```
SecAction id:9000,phase:5,pass,nolog,ctl:auditLogParts=ABCDFGH
```

You can then decide on your own whether the logging of part `E` is justified. If you're using full audit logging in particular, you'll 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" \
    id:9000,phase:5,pass,nolog,ctl:auditLogParts=+E
```

Removing Sensitive Data from Audit Logs

Most web application programmers are taught always to use POST methods for transactions that contain sensitive data. After all, it's well known that request bodies are never logged, meaning that sensitive data will never be logged, either. ModSecurity changes this situation, because it allows for full transaction logging. To deal with sensitive data that may find its way into the logs, ModSecurity uses the `sanitiseArg`, `sanitiseRequestHeader`, `sanitiseResponseHeader`, `sanitiseMatched`, and `sanitiseMatchedBytes` sanitization actions. 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 invoke them unconditionally with `SecAction`. Sanitization 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 for which you know the names. For example, let's assume that you have an application that uses the `password`, `oldPassword`, and `newPassword` parameters to transmit, well, passwords. This is what you'll do:

```
SecAction "id:9000,phase:5,pass,nolog,\
    sanitiseArg:password,\
    sanitiseArg:oldPassword,\
    sanitiseArg:newPassword"
```

Similarly, use `sanitiseRequestHeader` and `sanitiseResponseHeader` to remove the contents of the headers for which you know the names. For example, if you have an application that uses HTTP basic authentication, you'll need the following rule to prevent passwords from being logged:

```
SecAction "id:9000,phase:5,pass,nolog,\
    sanitiseRequestHeader:Authorization"
```

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

```
SecRule ARGS_NAMES "@rx password" \
    "id:9000,phase:5,pass,nolog,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}" \
    "id:9000,phase:5,pass,nolog,sanitiseMatched"
```

Finally, the `sanitiseMatchedBytes` action can remove the parts of input that contain sensitive information only, rather than 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}" \
    "id:9000,phase:5,pass,capture,nolog,sanitiseMatchedBytes"
```

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

```
SecRule ARGS "@verifyCC \d{13,16}" \
    "id:9000,phase:5,pass,capture,nolog,sanitiseMatchedBytes:4/4"
```

> **Warning**
>
> The sanitization actions work only for the data recorded in the audit log. However, sensitive data could end up in the error log if it's involved in a rule match, depending on the logging configuration of the rule. Likewise, sanitization doesn't work with XML and JSON request bodies.

Selective Audit Logging

Although full HTTP transaction logging sounds good in theory, in practice it's very difficult to use, because it slows down your server and requires large amounts of storage space. There are ways to get some of the same benefits for a fraction of the cost by using partial logging on demand.

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

To use this functionality, 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 id:1000,phase:1,pass,nolog,initcol:ip=%{REMOTE_ADDR}
```

Now, you need to add a rule that will trigger logging when one of the rules in the ruleset matches. We assume that all the rules in the ruleset assign a severity to their alerts. The default value for `HIGHEST_SEVERITY` is 255. Any value below 255 thus indicates that an alert has occurred; in practice, you might choose one of the real severity values, matching only on serious events. The following rule will start logging everything coming from an IP address after such a single rule match; to achieve that, we set the flag `ip.logflag` for up to one hour (3,600 seconds):

```
SecRule HIGHEST_SEVERITY "@lt 4" \
    id:9000,phase:5,pass,nolog,setvar:ip.logflag=1,expirevar:ip.logflag=3600
```

Finally, we add a rule that detects the flag and forces logging for all the requests from the flagged IP address:

```
SecRule IP:logflag "@gt 0" \
    id:9001,phase:5,pass,nolog,ctl:auditEngine=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 a higher-level discussion of how to place ModSecurity configuration within Apache's own directives. Finally, the interesting bits are here!

Index

R

T

V

W