# Apache Security with mod\_security 6.0

So for this chapter we are going to try and secure our apache web server to the best of our abilities. For this chapter we will add a layer of security by using a module that focuses on security. Naturally this module is called **mod\_security**. Now, what is mod\_security exactly? Mod\_security is a toolkit for real-time **web application monitoring, logging and access control**. **Mod\_security** is very flexible and lets the user choose with features to implement. Think of it as being a programming language to programmers(Just concept, not an actuall language, although it has its on syntax and engine) . Programmers don't have to implement every aspect of a computer programming language to write specific programs that performs a specific tasks, the programmers simply chooses which features of the programming language to implement. In short, you can chose which features of mod\_security to implement.

I stress this because it took me a while to wrap my head around what mod\_security actually is and its purpose. Let's look at some features.

Taken directly from <https://modsecurity.org/about.html> 2/28/17

* **Real-time application security:** monitoring and access control At its core, ModSecurity gives you access to the **HTTP traffic stream, in real-time, along with the ability to inspect it.** This is enough for real-time security monitoring. There's an added dimension of what's possible through ModSecurity's persistent storage mechanism, which enables you to track system elements over time and perform event correlation. You are able to reliably block, if you so wish, because ModSecurity uses full request and response buffering.
* **Full HTTP traffic logging :**Web servers traditionally do very little when it comes to logging for security purposes. They log very little by default, and even with a lot of tweaking you are not able to get everything that you need. I have yet to encounter a web server that is able to log full transaction data. ModSecurity gives you that ability to log anything you need, including raw transaction data, which is essential for forensics. In addition, you get to choose which transactions are logged, which parts of a transaction are logged, and which parts are sanitized.
* **Continuous passive security assessment :** Security assessment is largely seen as an active scheduled event, in which an independent team is sourced to try to perform a simulated attack. Continuous passive security assessment is a variation of real-time monitoring, where, instead of focusing on the behavior of the external parties, you focus on the behavior of the system itself. It's an early warning system of sorts that can detect traces of many abnormalities and security weaknesses before they are exploited.
* **Web application hardening:** One of my favorite uses for ModSecurity is attack surface reduction, in which you selectively narrow down the HTTP features you are willing to accept (e.g., request methods, request headers, content types, etc.). ModSecurity can assist you in enforcing many similar restrictions, either directly, or through collaboration with other Apache modules. They all fall under web application hardening. For example, it is possible to fix many session management issues, as well as cross-site request forgery vulnerabilities.
* **Something small, yet very important to you:** Real life often throws unusual demands to us, and that is when the flexibility of ModSecurity comes in handy where you need it the most. It may be a security need, but it may also be something completely different. For example, some people use ModSecurity as an XML web service router, combining its ability to parse XML and apply XPath expressions with its ability to proxy requests. Who knew?

Since we know the primary uses for mod\_security its important to know that there are multiple ways to implement mod\_security.

**Embedded:** Because ModSecurity is an Apache module, you can add it to any compatible version of Apache. At the moment that means a reasonably recent Apache version from the 2.0.x branch, although a newer 2.2.x version is recommended. The embedded option is a great choice for those who already have their architecture laid out and don't want to change it. Embedded deployment is also the only option if you need to protect hundreds of web servers. In such situations, it is impractical to build a separate proxybased security layer. Embedded ModSecurity not only does not introduce new points of failure, but it scales seamlessly as the underlying web infrastructure scales. The main challenge with embedded deployment is that server resources are shared between the web server and ModSecurity.

* In very simple terms, deploying an instance of an embedded mod\_security, the user simply installs the module and loads it with apache.

**Reverse proxy:**

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

For our case and due to being on local machines, we are going to deploy mod\_security on an embedded server. Im going to assume you already have apache2 installed on a debian/ubuntu based host. So we will go directly into installing and configuring mod\_secruity.

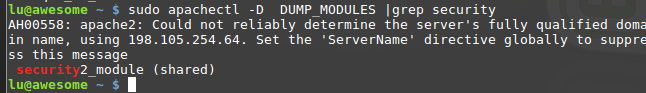
NOTE: if you dont know how to install apache or do not remember how the server works internally, refer back to chapter 5.

Lets update our repos and install mod\_security with one command.

[lu@awesome]$ sudo apt-get update && sudo apt-get install libapache2-mod-security2

At this point mod\_security should be installed and enabled by default. If for some reason later on in the chapter you experience issues. Check and make sure that mod\_security is even loaded.

[lu@awesome]$ sudo apachectl -D DUMP\_MODULES |grep security



You should see the about security2\_modlue (shared) if mod\_security is enabled.

NOTE: security is highlighted in red because of the pipe to |grep.

If mod\_security is not enabled by default use the a2enmod and a2dismod to enable and disable mod\_security.

* a2enmod = apache2 enable module
* a2dismod = apache2 disable module

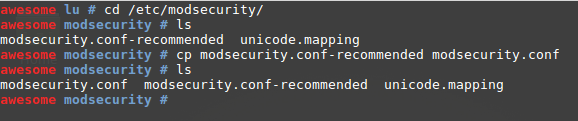
[lu@awesome]$ sudo a2enmod security2

[lu@awesome]$ sudo a2dismod security2



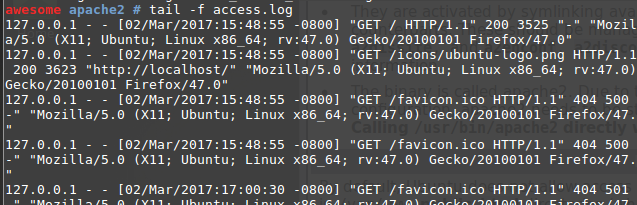
Now that mod\_security is enabled and installed, there are a few things we need to configure before we get to start using it. By default mod\_security comes with a pre-loaded / pre-configured configuration file that must be renamed in order to function properly. Let's simply rename the file with the mv file. This is the primary configuration file we will be working with.

NOTE: this file is recommended for beginners, you can always make your own configuration file from scratch.



NOTE: you need SUDO or ROOT access to modify.

The reason why we are implementing mod\_security is because default apache logging is not very helpful. So let's look at the contents of the debian based apache logs. So far the only log files available are 

* **Access.log** : This file will record any request made to the apache server. (GET and POST) . we can see the real time requests made to the apache server by tailing the file with the -f flag. 
  + Here i can see some basic information about the host connecting to the apache server. In this case, it is me with the localhost address of 127.0.0.1 and I am using a firefox browser.
* **error.log :** any internal server error apache encounters will be monitored in here.
* **Other\_vhosts\_access.log:** is used for

Since we made a copy of the main configuration file to an actual main configuration file we can go ahead and restart the apache2 server so that mod\_security can be good to go.

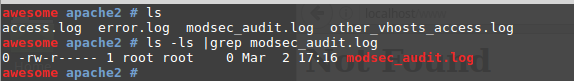
One of these commands is bound to work.

[lu@awesome]$ sudo apachectl restart

[lu@awesome]$ sudo service apache2 restart

[lu@awesome]$ sudo systemctl restart apache2.service

Now if we look into our /var/log/apache2/ folder. We should now see a modsec\_audit.log. If you see it then, congratulations you have a barebone mod\_security implementation!



If you take a look into the modsec\_audit.log and start making request to your browser, you will notice that its not logging anything, that is because mod\_security needs **rules** added to its configuration. This is where the whole “think of mod\_security as a programming language” comes in, we get to decide which features of mod\_security’s get implemented.

NOTE: mod\_security will not do ANYTHING unless it is specified. Don't expect any features to be implemented with others. You strictly have to tell it what to do. It's a dumb engine.

Here are some of the main Directives(rules) of mod\_security

### Directive Description

**SecArgumentSeparator**: Sets the application/x-www-form-urlencoded parameter separator

**SecCookieFormat** Sets the cookie parser version

**SecDataDir** Sets the folder for persistent storage

**SecRequestBodyAccess** Controls request body buffering

**SecRequestBodyInMemoryLimit** Sets the size of the per-request memory buffer

**SecRequestBodyLimit**  Sets the maximum request body size ModSecurity will accept

**SecRequestBodyLimitAction** Controls what happens once the request body limit is reached

**SecRequestBodyNoFilesLimit** Sets the maximum request body size, excluding uploaded files

**SecResponseBodyAccess** Controls response body buffering

**SecResponseBodyLimit** Specifies the response body buffering limit

**SecResponseBodyLimitAction** Controls what happens once the response body limit is reached

**SecResponseBodyMimeType** Specifies a list of response body MIME types to inspect

**SecResponseBodyMimeTypes** Clears the list of response body MIME types

**SecRuleEngine** Controls the operation of the rule engine

**SecTmpDir**  Sets the folder for temporary files

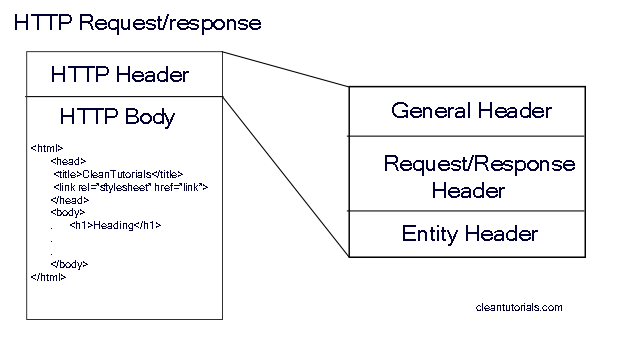
Don’t freak out if you don't know what these rules are yet. They deal with a lot of internal apache and HTTP workings. But for the sake of this chapter we will focus on the following. And also note that these rules will be present in the default configuration file we copied

**SecRequestBodyAccess:**

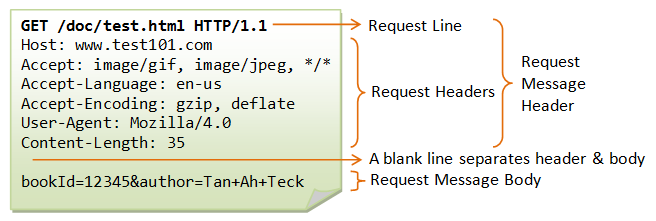
* **# Allow ModSecurity to access request bodies. If you don't,  
  # ModSecurity won't be able to see any POST parameters  
  # and that's generally not what you want.  
  SecRequestBodyAccess On**

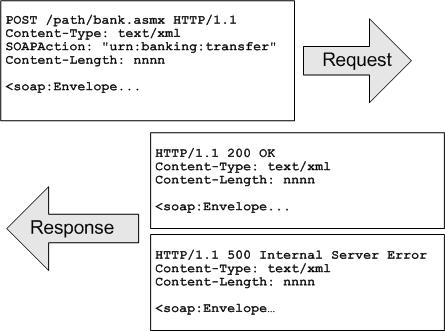
**WHAT ARE REQUEST BODIES?**  It's apart of the HTTP protocol. Now I don't have 20 hours to explain HTTP and how it works so i will briefly explain what we need to know for mod\_security. To find out how HTTP works visit this link <https://www.w3.org/Protocols/rfc2616/rfc2616.html>

Now, for every HTTP request a browser makes( when your connect to a website ) there are two parts to a complete HTTP request transaction .. In short it looks like this



To be in more detail. The http request/response looks like this. If this looks familiar it should! When we went over our apache logs. We saw this exact thing being logged into the access.log!





So by allowing mod\_security to ***SecRequestBodyAccess On*** we are allowing mod\_security to access the message body of a transaction! See that was not so bad.

Because mod\_security is allowed to access request bodies, it WILL BUFFER THEM. in other words it will store the information in RAM, this can cause an issue if you do not limit the size of the request bodies limit. Meaning if you dont set a maximum buffer(RAM) size for the request bodies, you can eventually dos attack yourself and cause your system to reboot if the ram fills up. We can set limiti sizes like this.

There are three directives that control how buffering is done. The first two, SecRequestBodyLimit and SecRequestBodyNoFilesLimit, establish request limits:

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

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

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

So as stated above the HTTP protocol can be either a request or response transaction. So a lot like request bodies, response bodies are the same concept. But in this case. If we are getting a response body from a website, we are essentially grabbing a web page. A webpage that could be very big in size and this can cause a heavy load on RAM and CPU. a request body is very small compared to a response body.

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

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

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

#MIME MultiPurpose Internet Mail Extensions   
SecResponseBodyMimeType text/plain text/html

This rule will essentially grab only the plain text of a response body. It will leave out any static files such as images or scripts. To read more about MIME visti this link <https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types>

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

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

However this can be bad if the repsonse page is larger that 512KB it will break. So there is a solution to this.

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

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

ModSecurity also lets us determine where uploaded files will be stored and the limitations we can set on file size.

# The location where ModSecurity will store intercepted  
# uploaded files. This location must be private to ModSecurity.  
SecUploadDir /opt/modsecurity/var/upload/  
  
# By default, do not intercept (nor store) uploaded files.  
SecUploadKeepFiles Off

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

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

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

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

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

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

# Log only what is really necessary.

SecAuditEngine RelevantOnly

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

# Also log requests that cause a server error.

SecAuditLogRelevantStatus ^5

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

# Log everything we know about a transaction.

SecAuditLogParts ABDEFHIJKZ

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

# Use a single file for logging.

SecAuditLogType Serial

SecAuditLog /opt/modsecurity/var/log/audit.log

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

# Specify the path for concurrent audit logging.

SecAuditLogStorageDir /opt/modsecurity/var/audit/

To Be Conitnued…