How To Set Up mod_security with Apache on Debian/Ubuntu

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=188*

Prelude

Mod security is a free Web Application Firewall (WAF) that works with Apache, Nginx and IIS. It supports a flexible rule engine to perform simple and complex operations and comes with a Core Rule Set (CRS) which has rules for SQL injection, cross site scripting, Trojans, bad user agents, session hijacking and a lot of other exploits. For Apache, it is an additional module which makes it easy to install and configure.

In order to complete this tutorial, you will need LAMP installed on your server.

Installing mod_security

Modsecurity is available in the Debian/Ubuntu repository:

```
apt-get install libapache2-modsecurity
```

Verify if the mod_security module was loaded.

```
apachectl -M | grep --color security
```

You should see a module named `security2_module (shared)` which indicates that the module was loaded.

Modsecurity's installation includes a recommended configuration file which has to be renamed:

```
mv /etc/modsecurity/modsecurity.conf{-recommended,}
```

Reload Apache

```
service apache2 reload
```

You'll find a new log file for mod_security in the Apache log directory:

```
root@panel:~# ls -l /var/log/apache2/modsec_audit.log
-rw-r----- 1 root root 0 Oct 19 08:08 /var/log/apache2/modsec_audit.log
```

Configuring mod_security

Out of the box, modsecurity doesn't do anything as it needs rules to work. The default configuration file is set to **DetectionOnly** which logs requests according to rule matches and doesn't block anything. This can be changed by editing the `modsecurity.conf` file:

```
nano /etc/modsecurity/modsecurity.conf
```

Find this line

```
SecRuleEngine DetectionOnly
```

and change it to:

```
SecRuleEngine On
```

If you're trying this out on a production server, change this directive only after testing all your rules.

Another directive to modify is `SecResponseBodyAccess`. This configures whether response bodies are buffered (i.e. read by modsecurity). This is only neccessary if data leakage detection and protection is required. Therefore, leaving it *On will use up panel resources and also increase the logfile size.*

Find this

```
SecResponseBodyAccess On
```

and change it to:

```
SecResponseBodyAccess Off
```

Now we'll limit the maximum data that can be posted to your web application. Two directives configure these:

```
SecRequestBodyLimit
SecRequestBodyNoFilesLimit
```

The `SecRequestBodyLimit` directive specifies the maximum POST data size. If anything larger is sent by a client the server will respond with a 413 Request Entity Too Large error. If your web application doesn't have any file uploads this value can be greatly reduced.

The value mentioned in the configuration file is

```
SecRequestBodyLimit 13107200
```

which is 12.5MB.

Similar to this is the `SecRequestBodyNoFilesLimit` directive. The only difference is that this directive limits the size of POST data minus file uploads-- this value should be "as low as practical."

The value in the configuration file is

```
SecRequestBodyNoFilesLimit 131072
```

which is 128KB.

Along the lines of these directives is another one which affects server performance: `SecRequestBodyInMemoryLimit`. This directive is pretty much self-explanatory; it specifies how much of "request body" data (POSTed data) should be kept in the memory (RAM), anything more will be placed in the hard disk (just like swapping). Since server instance use SSDs, this is not much of an issue; however, this can be set a decent value if you have RAM to spare.

```
SecRequestBodyInMemoryLimit 131072
```

This is the value (128KB) specified in the configuration file.

Testing SQL Injection

Before going ahead with configuring rules, we will create a PHP script which is vulnerable to SQL injection and try it out. Please note that this is just a basic PHP login script with no session handling. Be sure to change the MySQL password in the script below so that it will connect to the database:

```
/var/www/login.php

<html>
<body>
<?php
    if(isset($_POST['login']))
    {
        $username = $_POST['username'];
        $password = $_POST['password'];
        $con = mysqli_connect('localhost','root','password','sample');
        $result = mysqli_query($con, "SELECT * FROM `users` WHERE
username='$username' AND password='$password'");
        if(mysqli_num_rows($result) == 0)
            echo 'Invalid username or password';
        else
            echo '<h1>Logged in</h1><p>A Secret for you....</p>';
    }
    else
    {
?>
        <form action="" method="post">
            Username: <input type="text" name="username"/><br />
            Password: <input type="password" name="password"/><br />
            <input type="submit" name="login" value="Login"/>
        </form>
<?php
    }
?>
</body>
</html>
```

This script will display a login form. Entering the right credentials will display a message "A Secret for you."

We need credentials in the database. Create a MySQL database and a table, then insert usernames and passwords.

```
mysql -u root -p
```

This will take you to the `mysql>` prompt

```
create database sample;
connect sample;
```

```
create table users(username VARCHAR(100),password VARCHAR(100));
insert into users values('jesin','pwd');
insert into users values('alice','secret');
quit;
```

Open your browser, navigate to `http://yourwebsite.com/login.php` and enter the right pair of credentials.

```
Username: jesin
Password: pwd
```

You'll see a message that indicates successful login. Now come back and enter a wrong pair of credentials-- you'll see the message Invalid username or password.

We can confirm that the script works right. The next job is to try our hand with SQL injection to bypass the login page. Enter the following for the username field:

```
' or true --
```

Note that there should be a space after `--` this injection won't work without that space. Leave the **password field empty and hit the login button.**

Voila! The script shows the message meant for authenticated users.

Setting Up Rules

To make your life easier, there are a lot of rules which are already installed along with mod_security. These are called CRS (Core Rule Set) and are located in

```
root@cpanel:~# ls -l /usr/share/modsecurity-crs/
total 40
drwxr-xr-x 2 root root  4096 Oct 20 09:45 activated_rules
drwxr-xr-x 2 root root  4096 Oct 20 09:45 base_rules
drwxr-xr-x 2 root root  4096 Oct 20 09:45 experimental_rules
drwxr-xr-x 2 root root  4096 Oct 20 09:45 lua
-rw-r--r-- 1 root root 13544 Jul  2  2012 modsecurity_crs_10_setup.conf
drwxr-xr-x 2 root root  4096 Oct 20 09:45 optional_rules
drwxr-xr-x 3 root root  4096 Oct 20 09:45 util
```

The documentation is available at

```
root@cpanel1:~# ls -l /usr/share/doc/modsecurity-crs/
total 40
-rw-r--r-- 1 root root   469 Jul  2  2012 changelog.Debian.gz
-rw-r--r-- 1 root root 12387 Jun 18  2012 changelog.gz
-rw-r--r-- 1 root root  1297 Jul  2  2012 copyright
drwxr-xr-x 3 root root  4096 Oct 20 09:45 examples
-rw-r--r-- 1 root root  1138 Mar 16  2012 README.Debian
-rw-r--r-- 1 root root  6495 Mar 16  2012 README.gz
```

To load these rules, we need to tell Apache to look into these directories. Edit the `mod-security.conf` file.

```
nano /etc/apache2/mods-enabled/mod-security.conf
```

Add the following directives inside `<IfModule security2_module> </IfModule>`:

```
Include "/usr/share/modsecurity-crs/*.conf"
Include "/usr/share/modsecurity-crs/activated_rules/*.conf"
```

The `activated_rules` directory is similar to Apache's `mods-enabled` directory. The rules are available in directories:

```
/usr/share/modsecurity-crs/base_rules
/usr/share/modsecurity-crs/optional_rules
/usr/share/modsecurity-crs/experimental_rules
```

Symlinks must be created inside the `activated_rules` directory to activate these. Let us activate the SQL injection rules.

```
cd /usr/share/modsecurity-crs/activated_rules/
ln -s



/usr/share/modsecurity-crs/base_rules/modsecurity_crs_41_sql_injection_attacks.conf
.
```

Apache has to be reloaded for the rules to take effect.

```
service apache2 reload
```

Now open the login page we created earlier and try using the SQL injection query on the username field. If you had changed the `SecRuleEngine` directive to **On, you'll see a 403 Forbidden error. If it was left to the DetectionOnly option, the injection will be successful but the attempt would be logged in the `modsec_audit.log` file.**

Writing Your Own mod_security Rules

In this section, we'll create a rule chain which blocks the request if certain "spammy" words are entered in a HTML form. First, we'll create a PHP script which gets the input from a textbox and displays it back to the user.

```
/var/www/form.php
```

```
<html>
    <body>
        <?php
            if(isset($_POST['data']))
                echo $_POST['data'];
            else
            {
        ?>
                <form method="post" action="">
                        Enter something here:<textarea name="data"></textarea>
                        <input type="submit"/>
                </form>
        <?php
            }
        ?>
    </body>
</html>
```

Custom rules can be added to any of the configuration files or placed in modsecurity directories. We'll place our rules in a separate new file.

```
nano /etc/modsecurity/modsecurity_custom_rules.conf
```

Add the following to this file:

```
SecRule REQUEST_FILENAME "form.php" "id:'400001',chain,deny,log,msg:'Spam
detected'"
SecRule REQUEST_METHOD "POST" chain
SecRule REQUEST_BODY "@rx (?i:(pills|insurance|rolex))"
```

Save the file and reload Apache. Open `http://yourwebsite.com/form.php` in the browser and enter text containing any of these words: pills, insurance, rolex.

You'll either see a 403 page and a log entry or only a log entry based on `SecRuleEngine` setting. The syntax for **SecRule is**

```
SecRule VARIABLES OPERATOR [ACTIONS]
```

Here we used the chain* action to match variables REQUEST_FILENAME with form.php, REQUEST_METHOD with POST and REQUEST_BODY with the regular expression (@rx) string (pills|insurance|rolex). The ?i: does a case insensitive match. On a successful match of all these three rules, the *ACTION is to deny and log with the msg "Spam detected." The chain action simulates the logical AND to match all the three rules.

Excluding Hosts and Directories

Sometimes it makes sense to exclude a particular directory or a domain name if it is running an application like phpMyAdmin as modsecurity and will block SQL queries. It is also better to exclude admin backends of CMS applications like WordPress.

To disable modsecurity for a complete VirtualHost place the following

```
<IfModule security2_module>
    SecRuleEngine Off
</IfModule>
```

inside the `<VirtualHost>` section.

For a particular directory:

```
<Directory "/var/www/wp-admin">
    <IfModule security2_module>
```

```
        SecRuleEngine Off
    </IfModule>
</Directory>
```

If you don't want to completely disable modsecurity, use the `SecRuleRemoveById` directive to remove a particular rule or rule chain by specifying its ID.

```
<LocationMatch "/wp-admin/update.php">
    <IfModule security2_module>
        SecRuleRemoveById 981173
    </IfModule>
</LocationMatch>
```