Better protection with Apache's ModSecurity module

WATCHDOG

The Apache ModSecurity module provides extra protection for your web server. We'll show you why this optional application firewall is quickly becoming a favorite of webmasters and security experts.

BY HANNES KASPARICK

Most commercial web servers are devoted to the purpose of serving up dynamic, script-generated content in a reliable way. The very nature of the Internet means that unkown visitors from anywhere in the world may pay a visit to the site. Unfortunately, this all-important interaction between visitors and the web application opens up an attack vector. A skillful attacker could use a vulnerability to gain unauthorized access to the web server, and, once inside, the intruder could employ adidtional tools and tricks to do things that developers or webmasters never intended. The potential for damage is enormous, ranging from exposed contents of confidential files to a complete root compromise.

Cleanly programmed web applications provide one meaningful approach to preventing this kind of abuse, but the path is fraught with difficulty. Even the most experienced programmers are caught out from time to time, as vulnerabilities in established web applications just go to show.

Apache's ModSecurity module [1] provides validation and enhanced protection (Figure 1). The module is basically a web application firewall available both as an Apache module and as a standalone application (GPL with optional commercial support). The module validates incoming requests before passing them to the appropriate scripts based on the rules specified in the rule set. Just like the pattern files used by virus scanners, the rules comprise the signatures of typical attack techniques.

If a request matches one of the signatures, the mechanisms specified by action rules are enforced. This can include blocking the request, or forwarding the request to another web page. The module provides effective protection against attacks such as OS Command Injection, XSS/Cross-Site Scripting, and SQL Injection.

Danger from Outside

The main danger for web servers, and thus the administrator's main concern, is OS command injection attacks that execute system commands in the web server's context. In January 2005, for example, security experts discovered that the Awstats web statistics tool inadvertently allowed users to run arbitrary commands. The *awstats.pl?configdir=|echo;echo;ls%20-la%20%2F;id;echo;echo* string runs the *ls -la* command on the server with the permissions of Apache.

Giving attackers the ability to read confidential information stored on the web server can be similarly fatal. The exploit that targeted the popular Mambo CMS, which was disclosed in June 2005, passes in SQL statements in the URL, thus causing the web application to serve up a list of all user password hashes [7]. An attacker could then use the list to crack user passwords, possibly using Rainbow tables (pre-calculated hash tables). The following URL reveals the password entries:

```
http://server/mambo➥
/index.php?option=➥
```

Table 1: Rule Identifiers

ID	Use
0-99,999	#Local – for your own rules
100,000-199,999	#Reserved for ModSecurity
200,000-299,999	#Reserved for rules published on www.ModSecurity.org
300,000-;	Not assigned

```
com_content&task=vote&id=➥
%d&Itemid=%d&cid=1&user_rating=➥
1,rating_count=(SELECT/**/if➥
((ascii(substring((SELECT/**➥
/password/**/FROM/**/mos_users➥
/**/WHERE/**/id=%d),%d,1)))%s,➥
1145711457,0))[...]
```

The SQL statement embedded in this long string is: *SELECT password FROM mos_user WHERE id = User-ID*. The ModSecurity rule *SecFilterSelective ARGS "select. + from"* catches this attack by detecting the strings *select* and *from* as attack characteristics, and the attack fails (Figure 2). A would-be attacker simply gets to see the Apache 403 message *Forbidden: You don't have permission to access*.

Cross-site scripting attacks pose a greater threat to normal Internet users. Cross-site scripting involves the attacker attempting to run malevolent scripts on the victim's machine to access the victim's cookies, for example. In this light, it makes sense to filter *< script >* commands from *GET* and *POST* requests. As development of the ModSecurity module is progressing rapidly, administrators who plan to use ModSecurity are well advised to check out the homepage belonging to ModSecurity's author [1]. This is the place to go for updates on the ModSecurity Rule Sets project [2], for example.

Ready-to-run binary packages for ModSecurity are available for most distributions. Users with Debian can give the *aptitude install libapache2-mod-security* command to install the package on their disks; if you have FreeBSD, *pkg_add -r mod_security* will do the trick. After completing the install, enter *a2enmod mod-security* on Debian to load the module; FreeBSD does this automatically. This article is based on ModSecurity 1.9.2, although the popular 1.8.7 version has most of the features I refer to in this article.

Listing 1: Test rules

```
01 #SecFilterEngine On
02 #SecFilterSignatureAction
   log,deny,status:500
03 #SecFilter /bin/sh "id:1001,re
   v:2,severity:2,msg:'/bin/sh
   attack attempt'"
04 ## Regel für die Version 1.8:
05 ## SecFilter /bin/sh
```

First Run Out

The *[Fri Feb 24 11:55:12 2006] [notice] mod_security/1.9.2 configured* entry in my Apache *error.log* tells me that the module has installed successfully. To test whether the module is working, all I need is a simple rule set (Listing 1). The first line enables the filter engine; the second defines actions, and the third checks content for the strings that follow, */bin/sh* in this case. To keep things readable, it makes sense to store rule sets in separate files and use *include path to file* to bind them to your *apache2.conf*.

The extended parameters in the *Sec Filter* directive are new in version 1.9 and not supported by version 1.8. If you have the previous version, you will also need to replace *SecFilterSignatureAction* with *SecFilterDefaultAction*.

Calling the URL *http://Hostname/index.php?a=/bin/sh* in a web browser checks the test rule. If the rule works, the module will block access and display an *Internal Server Error* message. The *error.log* should contain the following:

```
[Fri Feb 24 14:25:12 2006] ➥
[error] [client 192.168.0.1] ➥
mod_security: Access ➥
denied with code 500.
Pattern match "/bin/sh" at ➥
REQUEST_URI [id "1001"] ➥
[rev "2"] [msg "/bin/sh attack➥
attempt"] [severity "2"] ➥
[hostname "192.168.0.20"] ➥
[uri"/modsec/index.php?a=➥
/bin/sh"]
```

Listing 2: Basic Configuration

```
01 ## Enable ModSecurity
02 #SecFilterEngine On
03 ## Log faulty requests and
   deny access
04 #SecFilterDefaultAction
   "deny,log,status:403"
05 ## Log errors only
06 ## SecFilterDefaultAction
   "pass,log"
07 #
08 ## Check POST data
09 #SecFilterScanPOST On
10 #
11 ## Check URL encoding
12 #SecFilterCheckURLEncoding On
13 #
14 ## Check Unicode encoding
15 #SecFilterCheckUnicodeEncoding
   On
16 #
17 ## Accept only Ascii
   characters 1 through 255
18 #SecFilterForceByteRange 1 255
19 #
20 ## Reduce server signature to
   a minimum
21 #SecServerSignature "Apache"
22 #
23 ## Log relevant data only
24 #SecAuditEngine RelevantOnly
25 #SecAuditLog /var/log/apache2/
   audit_log
26 #
27 ## Do not accept GET or HEAD
   requests in the body
28 #SecFilterSelective REQUEST_
   METHOD "^(GET|HEAD)$" chain
29 #SecFilterSelective HTTP_
   Content-Length "!^$"
30 #
31 ## Content length must be sent
   with each POST request
32 #SecFilterSelective REQUEST_
   METHOD "^POST$" chain
33 #SecFilterSelective HTTP_
   Content-Length "^$"
34 #
35 ## Discard unknown transfer
   encoding, with the exception
   of GET,
36 ## as this can cause problems
   with some clients
37 #SecFilterSelective REQUEST_
   METHOD "!^(GET|HEAD)$" chain
38 #SecFilterSelective HTTP_
   Content-Type \
39 #"!(^application/x-www-form-
urlencoded$|^multipart/
form-data;)"
40 #SecFilterSelective HTTP_
   Transfer-Encoding "!^$"
```

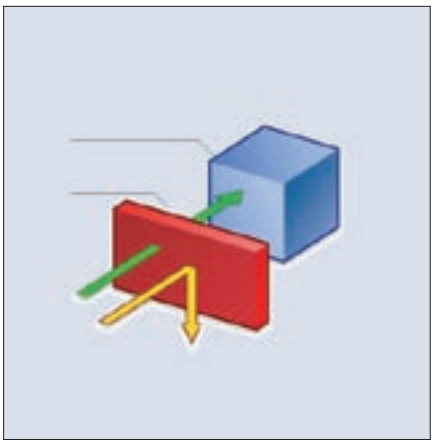

Figure 1: ModSecurity sits in front of the web server and the web applications, protecting both against attacks.

To make it easier for the system administrator to use scripts to evaluate the logfiles later on, each rule should have a unique ID: Table 1 gives you the reserved namespaces.

Coordinated Logging

The simplest form of logging merely writes entries to the Apache web server's *error.log*, although this option of writing to the error log makes analysis more difficult later.

As an alternative to the error log approach, you might like to consider the more extensive audit logging option shown in Figure 3. The Audit Log option was introduced with ModSecurity version 1.9. Among other things, the New Audit Log Type logging option supports logging of multiple events in separate logfiles, although it does necessitate enabling the Apache *mod_unique_id* module.

In addition to this, version 1.9 and newer support Guardian logging, that is, these versions can pass log data to HTTP Guardian [3] which in turn controls IPtables and pf firewalls, as well as the Snortsam [4] IDS controller. By default, HTTP Guardian blocks clients that send more than 120 requests per minute or more than 360 request in five minutes. The program is still at the development stage, but it works well and is well documented.

Basic Configuration

I will be using the basic configuration from Listing 2 as a starting point for some customization. Something similar to this listing is also available from the official project homepage. If you intend to test the rule set on a production system, it makes sense to just log potential infringements during the test phase to avoid blocking legitimate requests. You can use a *SecFilterDefaultAction* entry to do this, setting the parameters to *pass,log*.

Instead of returning a 403 error message when a rule is broken, ModSecurity will also let you point the offending request to any address or website using an entry such as *SecFilterDefaultAction "deny,log,redirect:http://targetpage.com"*. Filters help you define the criteria that ModSecurity applies to HTTP requests.

Filters always follow the pattern of *SecFilter SearchCriterion*, and as of version 1.9, the module has a few additional log parameters. ModSecurity distinguishes between three filter methods: simple (*SecFilter wget*), selective (*SecFilterSelective ARGS "union.+select"*) and output (*SecFilterSelective OUTPUT "Fatal error:" deny,status:500*).

A simple filter will always investigate the complete HTTP request, whereas a selective filter just investigates specific parts of the request.

Output filters scan the content served up by the web server, thus preventing it from being displayed if needed. An exclamation mark (*!*) will invert a filter rule. For example, *SecFilter !html* applies to any request that does not contain the *html* string.

The Dangers of Unicode

The Unicode standard provides a unified character set for international characters. The legacy ASCII character set uses only 7 or 8 bits to encode each character, thus restricting the number of characters to 128 or 256 respectively, and some of these characters are used for control purposes. Depending on the version, Unicode will use up to 32 bits (4 bytes) to encode each character. This means that Unicode can display runes and hieroglyphics.

On the downside, the technique of rewriting common exploits in Unicode has helped attackers bypass Intrusion Detection Systems (IDS) in the past. For example, the /character is represented as *&#47* in Unicode, and this kind of obfuscation might just keep the IDS at bay. ModSecurity decrypts Unicode strings by setting *SecFilterCheckUnicodeEncoding On*, giving filters downstream the ability to detect possible exploits.

Filters

The simplest form of a filter policy is *SecFilter SearchPattern*. The *SecFilter SearchPattern* filter policy tells ModSecurity to search all GET and POST requests for the pattern and trigger of the actions set by the default policy in case of a positive oucome.

Search keys can be simple expressions or regexes. Filter rules are not applied directly to the request but to a normalized copy. In other words, special (Unicode) encoded characters (see "The Dangers of Unicode") are decoded first, and any unnecessary / escaping is resolved. Null byte attacks targeted at vulnerabilities in server applications programmed in C or C++ are fended off by the *SecFilter hidden* filter.

The search patterns we have looked at thus far just check the whole HTTP request. This configuration could mean a bigger performance hit than you can accept. A *SecFilterSelective Location SearchPattern Action* entry lets you filter specific items. The location can be any CGI variable. The online documentation gives you the possible values and explains how to use them.

As an example of how to use *SecFilterSelective*, the following statement finds all access attempts that do not originate within the 192.168.0.0/?24 network: *SecFilterSelective "REMOTE_ADDR|REMOTE_HOST" !192.168.0.*

In combination with Apache 2, ModSecurity can filter the output from websites. If an attacker succeeds in injecting malicious SQL code that would output the *user_password* from a database, *SecFilterSelective OUTPUT "user_password" deny,status:500* would block the display. However, you will need to enable output filtering using *SecFilterScanOutput On* in order for this to happen.

Output filtering is disabled by default, and there is a good reason for this design: the resource overhead associated with using output filtering is considerable, as ModSecurity checks any content served up by the Apache server. Also, output filtering leads to the additional risk of inadvertently filtering legitimate content.

If you need to protect multiple virtual hosts that perform different tasks, the

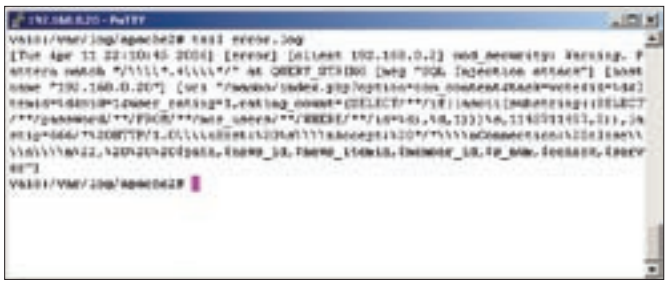

Figure 2: In our example, the Apache ModSecurity module successfully fends off an SQL Injection attack, recording the attempt in the web server's error.log.

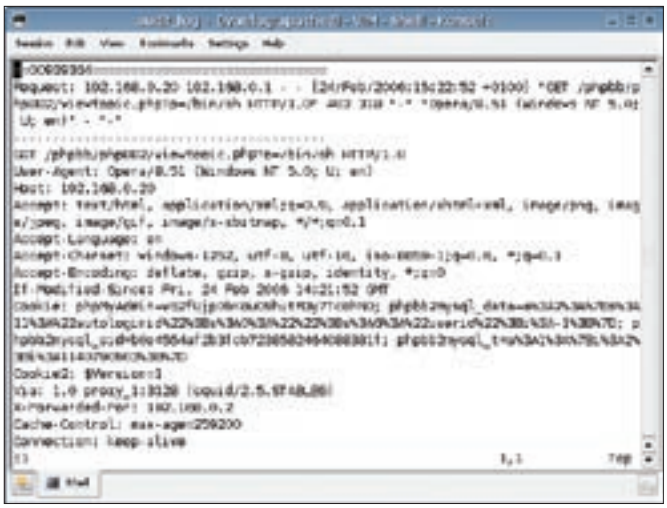

Figure 3: The ModSecurity audit log records details of the approach and circumstances surrounding the attack, thus giving the admin useful data concerning the day's events.

fact that ModSecurity supports rule inheritance via directories can be useful. Directory rules have precedence over global rules:

```
<Location /subcontext/>
   SecFilterRemove 1001
</Location>
```

This example simply disables rule ID 1001 while keeping the others. The following example does exactly the opposite – it disables all higher-level rules except for 1002 and 1003:

```
<Location /subcontext/>
   SecFilterInheritance Off
   SecFilterImport 1002 1003
</Location>
```

To make it easier to create rule sets, the Modescurity Rule Sets project [2] and the Gotroot [6] website offer pre-configured ModSecurity rule sets as downloads. The Gotroot rules support the new features in ModSecurity 1.9, which makes them incompatible with earlier versions.

Additional Features

ModSecurity has other security options besides simple filtering mechanisms. The *SecUploadApproveScript /path/to/script.sh* function lets you check file uploads for viruses by starting a script that triggers the virus scanner. The ModSecurity online documentation has a sample script. The module also sets up a chroot jail via the *SecChrootDir /var/www/* statement, thus preventing CGI scripts or binaries outside of the jail.

Performance

Depending on the scale of your rule set, ModSecurity can seriously affect your web server's responsiveness. A test using *ab* indicates the performance hit. The benchmark is part of the Apache package. We launched the tool with the following parameters: *time ab -n 500 -c 30 http://server/phbBB2/index.php* on a (not Centrino) Pentium 4 (1.8 GHz) CPU mobile system and measured a time of about 55 seconds for the benchmark on an Apache 2.0.55-4 without ModSecurity.

Enabling ModSecurity with the basic configuration shown in Listing 2 slows the Apache server down by about two percent, but after enabling the *modsecurity-general* rule set, as provided by the ModSecurity Rules project [6], the web server took about 15 to 20 percent longer to serve up the requested pages.

Administrators with heavily used web servers will need to keep the rule set as lean as possible to avoid major performance hits. Besides the number of rules, the complexity of the rules you use is an important factor. If you have filters with regular expressions, your rule set will consume far more CPU cycles than rules with simple comparative operators. As a general rule: the more precisely you customize your rule set to reflect your filtering needs, the less load it will generate.

As Apache 1 does not have an *PCRE regexp engine*, in contrast to Apache 2, the load on Apache version 1 is slightly higher. If a web server is exposed to a massive attack, a nicely tuned ModSecurity rule set can even improve your web server's performance, as the requests will not actually get through to your scripts.

As with any other rule-based security application, the potential security gain depends to a great extent on the rule set you use. In other words, attacks that your rule set does not explicitly cover will make it through to the server.

Conclusions

As an additional barrier to attacks on web applications, ModSecurity gives you extensive protection mechanisms. How effective these mechanisms are mainly depends on your rule set configuration, as is the case with all rule-based protection tools. Assuming a best-possible setup, the module can fend off most attacks on web servers and the web applications they host.

A rough and ready ModSecurity configuration could leave security holes and also prevent the Apache server from delivering legitimate content. When you are deploying ModSecurity, consider every filter rule carefully before you apply it. ■

INFO

[1] ModSecurity: *http://www.ModSecurity.org*

[2] ModSecurity rule sets: *http://www.ModSecurity.org/projects/rules/*

[3] Apache Httpd tools: *http://www.apachesecurity.net/tools/*

[4] Snortsam: *http://www.snortsam.net*

[5] Spread toolkit: *http://www.spread.org*

[6] ModSecurity rules: *http://www.gotroot.com/tiki-index.php?page=mod_security+rules*

[7] Mambo, SQL exploit: *http://www.milw0rm.com/exploits/1061*