Tutorial 6 - Embedding ModSecurity

What are we doing?

We are compiling the ModSecurity module, embedding it in the Apache web server, creating a base configuration and dealing with *false positives* for the first time.

Why are we doing this?

ModSecurity is a security module for the web server. The tool enables the inspection of both the request and the response according to predefined rules. This is also called the *Web Application Firewall*. It gives the administrator direct control over the requests and the responses passing through the system. The module also provides new options for monitoring, because the entire traffic between client and server can be written 1:1 to the hard disk. This helps in debugging. A *WAF* intervenes in HTTP traffic. This causes errors if a legitimate request is blocked. This is referred to as a *false positive*. Since the handling of these errors is an important part of working with *ModSecurity*, we will be starting off with this topic.

Requirements

- An Apache web server, ideally one created using the file structure shown in Tutorial 1 (Compiling an Apache web server).
- Understanding of the minimal configuration in Tutorial 2 (Configuring a minimal Apache server).
- An Apache web server with SSL/TLS support as in Tutorial 4 (Configuring an SSL server)
- An expanded access log and a set of shell aliases as discussed in Tutorial 5 (Extending and analyzing the access log)

Step 1: Downloading the source code and verifying the checksum

We previously downloaded the source code for the web server to */usr/src/apache*. We will now be doing the same with ModSecurity. To do so, as root we create the directory */usr/src/modsecurity/*, transfer it to ourselves and then download the code.

```
$> sudo mkdir /usr/src/modsecurity
$> sudo chown `whoami` /usr/src/modsecurity
$> cd /usr/src/modsecurity
$> wget https://www.modsecurity.org/tarball/2.9.0/modsecurity-2.9.0.tar.gz
```

Compressed, the source code is just over four megabytes in size. We now need to verify the checksum. It is provided in SHA256 format.

```
$> wget https://www.modsecurity.org/tarball/2.9.0/modsecurity-2.9.0.tar.gz.sha256
$> sha256sum --check modsecurity-2.9.0.tar.gz.sha256
```

We expect the following response:

```
modsecurity-2.9.0.tar.gz: OK
```

Step 2: Unpacking and configuring the compiler

We now unpack the source code and initiate the configuration. But before this it is essential to install three packages that constitute the prerequisite for compiling *ModSecurity*. A library for parsing XML structures and the base header files of the system's own Regular Expression Library: *libxml2-dev*, *libexpat1-dev* and *libpcre3-dev*.

The stage is thus set and we are ready for ModSecurity.

```
$> tar xvzf modsecurity-2.9.0.tar.gz
$> cd modsecurity-2.9.0
$> ./configure --with-apxs=/apache/bin/apxs \
```

```
--with-apr=/usr/local/apr/bin/apr-1-config \
--with-pcre=/usr/bin/pcre-config \
--enable-request-early
```

We created the */apache* symlink in the tutorial on compiling Apache. This again comes to our assistance, because independent from the Apache version being used, we can now have the ModSecurity configuration always work with the same parameters and always get access to the current Apache web server. The first two options establish the link to the Apache binary, since we have to make sure that ModSecurity is working with the right API version. The *with-pcre* option defines that we are using the system's own *PCRE-Library*, or Regular Expression Library, and not the one provided by Apache. This gives us a certain level of flexibility for updates, because we are becoming independent from Apache in this area, which has proven to work in practice. It requires the first installed *libpcre3-dev* package. The last option, *enable-request-early*, intervenes in the behavior of ModSecurity. This results in our security system handling the first processing phase after receiving the request header. We are thus not waiting for the finished request including body data, but can instead intervene immediately. This may be only a detail, but in practice it gives us a bit more control.

Step 3: Compiling

Following this preparation compiling should no longer pose a problem.

```
$> make
```

Step 4: Installing

Installation is also easily accomplished. Since we continue to be working on a test system, we transfer ownership of the installed module from the root user to ourselves, because for all of the Apache binaries we made sure to be the owner ourselves. This in turn produces a clean setup with uniform ownerships.

```
$> sudo make install
$> sudo chown `whoami` /apache/modules/mod_security2.so
```

The module has the number *2* in its name. This was introduced in the version jump to 2.0 when a reorientation of the module made this necessary. But this is only an minor detail.

Step 5: Creating the base configuration

We can now commence setting up a base configuration. ModSecurity is a module loaded by Apache. For this reason it is configured in the Apache configuration. Normally, it is proposed to configure ModSecurity in its own file and then to reload it as an *include*. We will however only be doing this with a part of the rules (in a subsequent tutorial). We will be pasting the base configuration into the Apache configuration to always keep it in view. In doing so, we will be expanding on our base Apache configuration. You can of course also combine this configuration using the *SSL setup* and the *application server setup*. For simplicity's sake we won't be doing the latter here. But we are embedding the extended log format that we became familiar with in the 5th tutorial. For this we are adding an additional, optional performance log that will help us find speed bottlenecks.

```
ServerName        localhost
ServerAdmin       root@localhost
ServerRoot        /apache
User              www-data
Group             www-data
PidFile           logs/httpd.pid

ServerTokens      Prod
UseCanonicalName  On
TraceEnable       Off

Timeout           10
MaxClients        100

Listen            127.0.0.1:80
Listen            127.0.0.1:443

LoadModule        mpm_event_module        modules/mod_mpm_event.so
```

```
LoadModule        unixd_module            modules/mod_unixd.so

LoadModule        log_config_module       modules/mod_log_config.so
LoadModule        logio_module            modules/mod_logio.so

LoadModule        authn_core_module       modules/mod_authn_core.so
LoadModule        authz_core_module       modules/mod_authz_core.so

LoadModule        ssl_module              modules/mod_ssl.so

LoadModule        unique_id_module        modules/mod_unique_id.so
LoadModule        security2_module        modules/mod_security2.so

ErrorLogFormat          "[%{cu}t] [%-m:%-l] %-a %-L %M"
LogFormat "%h %{GEOIP_COUNTRY_CODE}e %u [%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] \"%r\" %>s %b \
\"%{Referer}i\" \"%{User-Agent}i\" %v %A %p %R %{BALANCER_WORKER_ROUTE}e %X \"%{cookie}n\" \
%{UNIQUE_ID}e %{SSL_PROTOCOL}x %{SSL_CIPHER}x %I %O %{ratio}n%% \
%D %{ModSecTimeIn}e %{ApplicationTime}e %{ModSecTimeOut}e \
%{ModSecAnomalyScoreIn}e %{ModSecAnomalyScoreOut}e" extended

LogFormat "[%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] %{UNIQUE_ID}e %D \
PerfModSecInbound: %{TX.perf_modsecinbound}M \
PerfAppl: %{TX.perf_application}M \
PerfModSecOutbound: %{TX.perf_modsecoutbound}M \
TS-Phase1: %{TX.ModSecTimestamp1start}M-%{TX.ModSecTimestamp1end}M \
TS-Phase2: %{TX.ModSecTimestamp2start}M-%{TX.ModSecTimestamp2end}M \
TS-Phase3: %{TX.ModSecTimestamp3start}M-%{TX.ModSecTimestamp3end}M \
TS-Phase4: %{TX.ModSecTimestamp4start}M-%{TX.ModSecTimestamp4end}M \
TS-Phase5: %{TX.ModSecTimestamp5start}M-%{TX.ModSecTimestamp5end}M \
Perf-Phase1: %{PERF_PHASE1}M \
Perf-Phase2: %{PERF_PHASE2}M \
Perf-Phase3: %{PERF_PHASE3}M \
Perf-Phase4: %{PERF_PHASE4}M \
Perf-Phase5: %{PERF_PHASE5}M \
Perf-ReadingStorage: %{PERF_SREAD}M \
Perf-ReadingStorage: %{PERF_SWRITE}M \
Perf-GarbageCollection: %{PERF_GC}M \
Perf-ModSecLogging: %{PERF_LOGGING}M \
Perf-ModSecCombined: %{PERF_COMBINED}M" perflog

LogLevel                      debug
ErrorLog                      logs/error.log
CustomLog                     logs/access.log extended
CustomLog                     logs/modsec-perf.log perflog env=write_perflog

# == ModSec Base Configuration

SecRuleEngine                 On

SecRequestBodyAccess          On
SecRequestBodyLimit           10000000
SecRequestBodyNoFilesLimit    64000

SecResponseBodyAccess         On
SecResponseBodyLimit          10000000

SecPcreMatchLimit             15000
SecPcreMatchLimitRecursion    15000

SecTmpDir                     /tmp/
SecDataDir                    /tmp/
SecUploadDir                  /tmp/

SecDebugLog                   /apache/logs/modsec_debug.log
SecDebugLogLevel              0

SecAuditEngine                RelevantOnly
SecAuditLogRelevantStatus     "^(?:5|4(?!04))"
SecAuditLogParts              ABIJEFHKZ

SecAuditLogType               Concurrent
SecAuditLog                   /apache/logs/modsec_audit.log
SecAuditLogStorageDir         /apache/logs/audit/
```

```
SecDefaultAction              "phase:1,pass,log,tag:'Local Lab Service'"


# == ModSec Rule ID Namespace Definition
# Service-specific before Core-Rules:    10000 -  49999
# Service-specific after Core-Rules:     50000 -  79999
# Locally shared rules:                  80000 -  99999
#  - Performance:                        90000 -  90199
# Recommended ModSec Rules (few):       200000 - 200010
# OWASP Core-Rules:                     900000 - 999999


# === ModSec timestamps at the start of each phase (ids: 90000 - 90009)

SecAction "id:'90000',phase:1,nolog,pass,setvar:TX.ModSecTimestamp1start=%{DURATION}"
SecAction "id:'90001',phase:2,nolog,pass,setvar:TX.ModSecTimestamp2start=%{DURATION}"
SecAction "id:'90002',phase:3,nolog,pass,setvar:TX.ModSecTimestamp3start=%{DURATION}"
SecAction "id:'90003',phase:4,nolog,pass,setvar:TX.ModSecTimestamp4start=%{DURATION}"
SecAction "id:'90004',phase:5,nolog,pass,setvar:TX.ModSecTimestamp5start=%{DURATION}"

# SecRule REQUEST_FILENAME "@beginsWith /" "id:'90005',phase:5,t:none,nolog,noauditlog,pass,setenv:write_pe



# === ModSec Recommended Rules (in modsec src package) (ids: 200000-200010)

SecRule REQUEST_HEADERS:Content-Type "text/xml" \
  "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=XML"

SecRule REQBODY_ERROR "!@eq 0" \
  "id:'200001',phase:2,t:none,deny,status:400,log,msg:'Failed to parse request body.',\
logdata:'%{reqbody_error_msg}',severity:2"

SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"id:'200002',phase:2,t:none,log,deny,status:403, \
msg:'Multipart request body failed strict validation: \
PE %{REQBODY_PROCESSOR_ERROR}, \
BQ %{MULTIPART_BOUNDARY_QUOTED}, \
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
DB %{MULTIPART_DATA_BEFORE}, \
DA %{MULTIPART_DATA_AFTER}, \
HF %{MULTIPART_HEADER_FOLDING}, \
LF %{MULTIPART_LF_LINE}, \
SM %{MULTIPART_MISSING_SEMICOLON}, \
IQ %{MULTIPART_INVALID_QUOTING}, \
IP %{MULTIPART_INVALID_PART}, \
IH %{MULTIPART_INVALID_HEADER_FOLDING}, \
FL %{MULTIPART_FILE_LIMIT_EXCEEDED}'"

SecRule TX:/^MSC_/ "!@streq 0" \
  "id:'200004',phase:2,t:none,deny,status:500,msg:'ModSecurity internal error flagged: %{MATCHED_VAR_NAME}'


# === ModSecurity Rules
#
# ...


# === ModSec timestamps at the end of each phase (ids: 90010 - 90019)

SecAction "id:'90010',phase:1,pass,nolog,setvar:TX.ModSecTimestamp1end=%{DURATION}"
SecAction "id:'90011',phase:2,pass,nolog,setvar:TX.ModSecTimestamp2end=%{DURATION}"
SecAction "id:'90012',phase:3,pass,nolog,setvar:TX.ModSecTimestamp3end=%{DURATION}"
SecAction "id:'90013',phase:4,pass,nolog,setvar:TX.ModSecTimestamp4end=%{DURATION}"
SecAction "id:'90014',phase:5,pass,nolog,setvar:TX.ModSecTimestamp5end=%{DURATION}"


# === ModSec performance calculations and variable export (ids: 90100 - 90199)

SecAction "id:'90100',phase:5,pass,nolog,setvar:TX.perf_modsecinbound=%{PERF_PHASE1}"
SecAction "id:'90101',phase:5,pass,nolog,setvar:TX.perf_modsecinbound=+%{PERF_PHASE2}"
SecAction "id:'90102',phase:5,pass,nolog,setvar:TX.perf_application=%{TX.ModSecTimestamp3start}"
SecAction "id:'90103',phase:5,pass,nolog,setvar:TX.perf_application=-%{TX.ModSecTimestamp2end}"
SecAction "id:'90104',phase:5,pass,nolog,setvar:TX.perf_modsecoutbound=%{PERF_PHASE3}"
```

```
SecAction "id:'90105',phase:5,pass,nolog,setvar:TX.perf_modsecoutbound=+%{PERF_PHASE4}"
SecAction "id:'90106',phase:5,pass,nolog,setenv:ModSecTimeIn=%{TX.perf_modsecinbound}"
SecAction "id:'90107',phase:5,pass,nolog,setenv:ApplicationTime=%{TX.perf_application}"
SecAction "id:'90108',phase:5,pass,nolog,setenv:ModSecTimeOut=%{TX.perf_modsecoutbound}"
SecAction "id:'90109',phase:5,pass,nolog,setenv:ModSecAnomalyScoreIn=%{TX.inbound_anomaly_score}"
SecAction "id:'90110',phase:5,pass,nolog,setenv:ModSecAnomalyScoreOut=%{TX.outbound_anomaly_score}"



SSLCertificateKeyFile   /etc/ssl/private/ssl-cert-snakeoil.key
SSLCertificateFile      /etc/ssl/certs/ssl-cert-snakeoil.pem

SSLProtocol             All -SSLv2 -SSLv3
SSLCipherSuite          'kEECDH+ECDSA kEECDH kEDH HIGH +SHA !aNULL !eNULL !LOW !MEDIUM !MD5 !EXP !DSS \
!PSK !SRP !kECDH !CAMELLIA !RC4'
SSLHonorCipherOrder     On

SSLRandomSeed           startup file:/dev/urandom 2048
SSLRandomSeed           connect builtin

DocumentRoot            /apache/htdocs

<Directory />

        Require all denied

        Options SymLinksIfOwnerMatch
        AllowOverride None

</Directory>

<VirtualHost 127.0.0.1:80>

      <Directory /apache/htdocs>

        Require all granted

        Options None
        AllowOverride None

      </Directory>

</VirtualHost>

<VirtualHost 127.0.0.1:443>

      SSLEngine On

      <Directory /apache/htdocs>

              Require all granted

              Options None
              AllowOverride None

      </Directory>

</VirtualHost>
```

What has been added are the *mod_security2.so* and *mod_unique_id.so* modules and the additional performance log. At first we define the *LogFormat* and then a few lines below the *logs/modsec-perf.log* file. A condition is added to the end of this line: Only when the *write_perflog* environment variable is set will this log file actually be written. We can thus decide whether we need performance data or not for each request. This saves resources and gives us the option of working with pinpoint precision: We can thus include only specific paths in the log or concentrate on individual client IP addresses.

The ModSecurity base configuration begins on the next line: We define the base settings of the module in this part. Then in a separate part come individual security rules, most of which are a bit complicated. Let's go through this configuration step-by-step: *SecRuleEngine* is what enables ModSecurity in the first place. We then enable access to the request body and set two limits: By default only the header lines of the request are examined. This is like looking only at the envelope of a letter. Inspecting the body and thus the content of the request of course involves more work and takes more time, but a large

number of attacks are not detectable from outside, which is why we are enabling this. We then limit the size of the request body to 10 MB. This includes file uploads. For requests with body, but without file upload, such as an online form, we then specify 64 KB as the limit. In detail, *SecRequestBodyNoFilesLimit* is responsible for *Content-Type application/x-www-form-urlencoded*, while *SecRequestBodyLimit* takes care of *Content-Type: multipart/form-data*.

On the response side we enable body access and in turn define a limit of 10 MB. No differentiation is made here in the transfer of forms or files; all of them are files.

Now comes the memory reserved for the *PCRE library*. ModSecurity documentation suggests a value of 1500 bytes. But this quickly leads to problems in practice. Our base configuration with a limit of 15000 is a bit more robust. If problems still occur, values above 100000 are also manageable; memory requirements grow only marginally.

ModSecurity requires three directories for data storage. We put all of them in the *tmp directory*. For productive operation this is of course the wrong place, but for the first baby steps it's fine and it is not easy to give general recommendations for the right choice of this directory, because the local environment plays a big role. For the aforementioned directories this concerns temporary data, then about session data that should be retained after a server restart, and finally temporary storage for file uploads which during inspection should not use too much memory and above a specific size are stored on the hard disk.

ModSecurity has a very detailed *debug log*. The configurable log level ranges from 0 to 9. We leave it at 0 and are prepared to be able to increase it when problems occur in order to see exactly how the module is working. In addition to the actual *rule engine*, an *audit engine* also runs within ModSecurity. It organizes the logging of requests. Because in case of attack we would like to get as much information as possible. With *SecAuditEngine RelevantOnly* we define that only *relevant* requests should be logged. What's relevant to us is what we define on the next line via a regular expression: All requests whose HTTP status begins with 4 or 5, but not 404. At a later point in time we will see that other things can be defined as relevant, but this rough classification is good enough for the start. It then continues with a definition of the parts of this request that should be logged. We are already familiar with the request header (part B), the request body (part I), the response header (part F) and the response body (part E). Then comes additional information from ModSecurity (parts A, H, K, Z) and details about uploaded files, which we do not map completely (part J). A detailed explanation of these audit log parts are available in the ModSecurity reference manual.

Depending on request, a large volume of data is written to the audit log. There are often several hundred lines for each request. On a server under a heavy load with many simultaneous requests this can cause problems writing the file. This is why the *Concurrent Log Format* was introduced. It keeps a central audit log including the most important information. The detailed information in the parts just described are stored in in individual files. These files are placed in the directory tree defined using the *SecAuditLogStorageDir* directive. Every day, ModSecurity creates a directory in this tree and another directory for each minute of the day (however, only if a request was actually logged within this minute). In them are the individual requests with file names labeled by date, time and the unique ID of the request.

Here is an example from the central audit log:

```
localhost 127.0.0.1 - - [17/Oct/2015:15:54:54 +0200] "POST /index.html HTTP/1.1" 200 45 "-" "-" \
UYkHrn8AAQEAAHb-AM0AAAAB "-" /20130507/20130507-1554/20130507-155454-UYkHrn8AAQEAAHb-AM0AAAAB \
0 20343 md5:a395b35a53c836f14514b3fff7e45308
```

We see some information about the request, the HTTP status code and shortly afterward the *unique ID* of the request, which we also find in our access log. An absolute path follows a bit later. But it only appears to be absolute. Specifically, we have to add this part of the path to the value in *SecAuditLogStorageDir*. For us this means */apache/logs/audit/20130507/20130507-1554/20130507-155454-UYkHrn8AAQEAAHb-AM0AAAAB*. We can then find the details about the request in this file.

```
--5a70c866-A--
[17/Oct/2013:15:54:54 +0200] UYkHrn8AAQEAAHb-AM0AAAAB 127.0.0.1 42406 127.0.0.1 80
--5a70c866-B--
POST /index.html HTTP/1.1
User-Agent: curl/7.35.0 (x86_64-pc-linux-gnu) libcurl/7.35.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp
Accept: */*
Host: 127.0.0.1
Content-Length: 3
Content-Type: application/x-www-form-urlencoded

...
```

The parts described divide the file into sections. What follows is part *--5a70c866-A--* as part A, then *--5a70c866-B--* as part B,

etc. We will be having a look at this log in detail in a subsequent tutorial. This introduction should suffice for the moment. But what is not sufficient is our file system. Because, in order to write the *audit log* at all, the directory must first be created and the appropriate permissions assigned:

```
$> sudo mkdir /apache/logs/audit
$> sudo chown www-data:www-data /apache/logs/audit
```

This brings us to the *SecDefaultAction* directive. It denotes the basic setting of a security rule. Although we can define this value for each rule, it is normal to work with one default value which is then inherited by all of the rules. ModSecurity is aware of five phases. Phase 1 listed here starts once the request headers have arrived on the server. In compiling the module we used *--enable-request-early* to define that this really is the case, because for technical reasons this behavior was changed and phase 1 was consolidated with the following phase. We use the aforementioned flag to reverse this change in behavior. The other phases are the *request body phase (phase 2)*, *response header phase (phase 3)*, *response body phase (phase 4)* and the *logging phase (phase 5)*. We then say that when a rule takes effect we would normally like the request to pass. We will be defining blocking measures separately. We would like to log; meaning that we would like to see a message about the triggered rule in the Apache server's *error log* and ultimately assign each of these log entries a *tag*. The tag set, *Local Lab Service*, is only one example of the strings, even several of them, that can be set. In a larger company it can for example be useful for adding additional information about a service (contract number, customer contact details, references to documentation, etc.). This information is then included along with every log entry. This may first sound like a waste of resources, but one employee on an operational security team may be responsible for several hundred services and the URL alone is not enough at this time for unknown services. These service metadata, added by using tags, enable a quick and appropriate reaction to attacks.

This brings us to the ModSecurity rules. Although the module works with the limits defined above, the actual functionality lies mainly in the individual rules that can be expressed in their own rule language. But before we have a look at the individual rules, a comment section with definitions of the namespace of the rule ID numbers follows in the Apache configuration. Each ModSecurity rule has a number for identification. In order to keep the rules manageable, it is useful to cleanly divide up the namespace.

The OWASP ModSecurity Core Rule Set project provides a basic set of over 200 ModSecurity rules. We will be embedding these rules in the next tutorial. They have IDs beginning with the number 900,000 and range up to 999,999. For this reason, we shouldn't set up any rules in this range. The ModSecurity sample configuration provides a few rules in the range starting at 200,000. Our own rules are best organized in the big spaces in between. I suggest keeping in the range below 100,000.

If ModSecurity is being used for multiple services, eventually some shared rules will be used. These are self-written rules configured for each of their own instances. We put these in the 80,000 to 99,999 range. For the other service-specific rules it often plays a role as to whether they are defined before or after the core rules. For logical reasons, we therefore divide the remaining space into two sections: 10,000 to 49,999 for service-specific rules before the core rules and 50,000 to 79,999 after the core rules. Although we won’t yet be embedding the core rules in this tutorial, we will be preparing for them.

This brings us to the first rules. We start off with a block of performance data. There are not yet any security-related rules, but the definition of information for the path of the request within ModSecurity. We use the *SecAction* directive. A *SecAction* is always performed without condition. A comma separated list with instructions follows as parameters. We initially define the rule ID, then the phase in which the rule is to run (1 to 5). We do no wish to have an entry in the server’s error log (*nolog*). Furthermore, we let the request *pass* and set multiple internal variables: We define a timestamp for each ModSecurity phase. As it were, an intermediate time within the request when starting each individual phase. This is done by using the clock running in the form of the *Duration* variables which begin ticking in microseconds at the start of the request.

The rule with ID 90005 is commented out. We can enable it in order to set the Apache *write_perflog* environment variable. Once we do that the performance log defined in the Apache section will be written. This rule is no longer defined as *SecAction*, but as *SecRule*. A preceding condition is added to the rule instruction here. In our case we inspect *REQUEST_FILENAME* with respect to the beginning of the string. If the string begins with */*, then the subsequent instructions including setting the environment variables should be performed. Of course, every valid request URI begins with the */* character. But if we only want to enable the log for specific paths (e.g. */login*), we are then prepared for this and only need to modify the path.

So much for this performance part. Now come the rules proposed by the *ModSecurity* project in the sample configuration file. They have rule IDs starting at 200,000 and are not very numerous. The first rule inspects the *request headers Content-Type*. The rule applies when these headers match the text *text/xml*. It is evaluated in phase 1. After the phase comes the *t:none* instruction. This means *transformation: none*. We do not want to transform the parameters of the request prior to processing this rule. Following *t:none* a transformation with the self-explanatory name *t:lowercase* is applied to the text. Using *t:none* we

delete all predefined default transformations if need be and then execute *t:lowercase*. This means that we will be touching *text/xml*, *Text/Xml*, *TEXT/XML* and all other combinations in the *Content-Type* header. If this rule applies, then we perform a *control action* at the very end of the line: We choose *XML* as the processor of the *request body*. There is one detail still to be explained: The preceding commented out rule introduced the operator *@beginsWith*. By contrast, no operator is designated here. *Default-Operator @rx* is applied. This is an operator for regular expressions (*regex*). As expected, *beginsWith* is a very fast operator while working with regular expressions is cumbersome and slow.

By contrast, the next rule is a bit more complicated. We are inspecting the internal *REQBODY_ERROR* variable. In the condition part we use the numerical comparison operator *@eq*. The exclamation mark in front negates its value. The syntax thus means if the *REQBODY_ERROR* is not equal to zero. Of course, we could also work with a regular expression here, but the *@eq* operator is more efficient when being processed by the module. In the action part of the rule *deny* is applied for the first time. The request should thus be blocked if processing the request body resulted in an error. Specifically, we return HTTP status code *400 Bad Request* (*status:400*). We would like to log first and specify the message. As additional information we also write to a separate log field called *logdata* the exact description of the error. This information will appear in both the server's error log as well as in the audit log. Finally, the *severity* is assigned to the rule. This is the degree of importance for the rule, which can be used in evaluating very many rule violations.

The rule with the ID 200002 also deals with errors in the request body. This concerns *multipart HTTP bodies*. It applies if files are to be transferred to the server via HTTP requests. This is very useful on the one hand, but poses a big security problem on the other. This is why ModSecurity very precisely inspects *multipart HTTP bodies*. It has an internal variable called *MULTIPART_STRICT_ERROR*, which combines the numerous checks. If there is a value other than 0 here, then we block the request using status code 403 (*forbidden*). In the log message we then report the results of the individual checks. In practice you have to know that in very rare cases this rule may also be applied to legitimate requests. If this is the case, it may have to be modified or disabled as a *false positive*. We will be returning to the elimination of false positives further below and will become familiar with the topic in detail in a subsequent tutorial.

The ModSecurity distribution sample configuration has another rule with ID 200003. However, I have not included it in the tutorial, because in practice it blocks too many legitimate requests (*false positives*). The *MULTIPART_UNMATCHED_BOUNDARY* variable is checked. This value, which signifies an error in the boundary of multipart bodies, is prone to error and frequently reports text snippets which do not indicate boundaries. In my opinion, it has not shown itself to be useful in practice.

With 200004 comes another rule which intercepts internal processing errors. Unlike the preceding internal variables, here we are looking for a group of variables dynamically provided along with the current request. A data sheet called *TX* (transaction) is opened for each request. In ModSecurity jargon we refer to a *collection* of variables and values. While processing a request ModSecurity now in some circumstances sets additional values in the *TX collection*, in addition to the variables already inspected. The names of these variables begin with the prefix *MSC_*. We now access in parallel all variables of this pattern in the collection. This is done via the *TX:/^MSC_/* construct. Thus, the transaction collection and then variable names matching the regular expression *^MSC_*: A word beginning with *MSC_*. If one of these found variables is not equal to zero, we then block the request using HTTP status 500 (*internal server error*) and write the variable names in the log file.

We have now looked at a few rules and have become familiar with the principle functioning of the ModSecurity *WAF*. The rule language is demanding, but very systematic. The structure is unavoidably oriented to the structure of Apache directives. Because before ModSecurity is able to process the directives, they are read by Apache's configuration parser. This is also accompanied by complexity in the way they are expressed. *ModSecurity* is currently being developed in a direction making the module independent from Apache. We will hopefully be benefitting from a configuration that is easier to read.

Now comes a comment in the configuration file which marks the spot for additional rules to be entered. Following this block, which in some circumstances can become very large, come yet more rules that provide performance data for the performance log defined above. The block containing rule IDs 90010 to 90014 stores the time of the end of the individual ModSecurity phases. This corresponds to the 90000 - 90004 block of IDs we became familiar with above. Calculations with the performance data collected are then performed in the last ModSecurity block. For us this means that we totaling up the time that phase 1 and phase 2 need in the *perf_modsecinbound* variable. In the rule with ID 90100 this variable is set to the performance of phase 1, in the following rule the performance of phase 2 is added to it. We have to calculate the variable *perf_application* from the timestamps. To do this, we subtract the end of phase 2 from the start of phase 3 (rule IDs 90102 and 90103). This is of course not an exact calculation of the time that the application itself needs on the server, because other Apache modules play a role (such as authentication), but the value is an indication that sheds light on whether ModSecurity is actually limiting performance or whether the problem more likely lies with the application. Finally, in rule IDs 90104 and 90105 the calculation of the time used in phases 3 and 4, similar to phases 1 and 2. This gives us three relevant values which simply summarize performance: *perf_modsecinbound*, *perf_application* and *perf_modsecoutbound*. They appear in a separate performance log. We have, however, provided enough space for these three values in the normal access log. There we have

ModSecTimeIn, *ApplicationTime* and *ModSecTimeOut*. In rules 90106 to 90108 we export our *perf* values to the corresponding environment variables in order for them to appear in the *access log*. At the end come rules 90109 and 90110. In them we export the *OWASP ModSecurity Core Rules* anomaly values. These values are not yet written, but because we will be making these rules available in the next tutorial, we can already prepare for variable export here.

We are now at the point that we can understand the performance log. The definition above is accompanied by the following parts:

```
LogFormat "[%{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t] %{UNIQUE_ID}e %D \
PerfModSecInbound: %{TX.perf_modsecinbound}M \
PerfAppl: %{TX.perf_application}M \
PerfModSecOutbound: %{TX.perf_modsecoutbound}M \
TS-Phase1: %{TX.ModSecTimestamp1start}M-%{TX.ModSecTimestamp1end}M \
TS-Phase2: %{TX.ModSecTimestamp2start}M-%{TX.ModSecTimestamp2end}M \
TS-Phase3: %{TX.ModSecTimestamp3start}M-%{TX.ModSecTimestamp3end}M \
TS-Phase4: %{TX.ModSecTimestamp4start}M-%{TX.ModSecTimestamp4end}M \
TS-Phase5: %{TX.ModSecTimestamp5start}M-%{TX.ModSecTimestamp5end}M \
Perf-Phase1: %{PERF_PHASE1}M \
Perf-Phase2: %{PERF_PHASE2}M \
Perf-Phase3: %{PERF_PHASE3}M \
Perf-Phase4: %{PERF_PHASE4}M \
Perf-Phase5: %{PERF_PHASE5}M \
Perf-ReadingStorage: %{PERF_SREAD}M \
Perf-ReadingStorage: %{PERF_SWRITE}M \
Perf-GarbageCollection: %{PERF_GC}M \
Perf-ModSecLogging: %{PERF_LOGGING}M \
Perf-ModSecCombined: %{PERF_COMBINED}M" perflog
```

- %{%Y-%m-%d %H:%M:%S}t.%{usec_frac}t means, as in our normal log, the timestamp the request was received with a precision of microseconds.
- %{UNIQUE_ID}e : The unique ID of the request
- %D : The total duration of the request from receiving the request line to the end of the complete request in microseconds.
- PerfModSecInbound: %{TX.perf_modsecinbound}M : Summary of the time needed by ModSecurity for an inbound request.
- PerfAppl: %{TX.perf_application}M : Summary of the time used by the application
- PerfModSecOutbound: %{TX.perf_modsecoutbound}M : Summary of the time needed in ModSecurity to process the response
- TS-Phase1: %{TX.ModSecTimestamp1start}M-%{TX.ModSecTimestamp1end}M : The timestamps for the start and end of phase 1 (after receiving the request headers)
- TS-Phase2: %{TX.ModSecTimestamp2start}M-%{TX.ModSecTimestamp2end}M : The timestamps for the start and end of phase 2 (after receiving the request body)
- TS-Phase3: %{TX.ModSecTimestamp3start}M-%{TX.ModSecTimestamp3end}M : The timestamps for the start and end of phase 3 (after receiving the response headers)
- TS-Phase4: %{TX.ModSecTimestamp4start}M-%{TX.ModSecTimestamp4end}M : The timestamps for the start and end of phase 4 (after receiving the response body)
- TS-Phase5: %{TX.ModSecTimestamp5start}M-%{TX.ModSecTimestamp5end}M : The timestamps for the start and end of phase 5 (logging phase)
- Perf-Phase1: %{PERF_PHASE1}M : Calculation of the performance of the rules in phase 1 performed by ModSecurity
- Perf-Phase2: %{PERF_PHASE2}M : Calculation of the performance of the rules in phase 2 performed by ModSecurity
- Perf-Phase3: %{PERF_PHASE3}M : Calculation of the performance of the rules in phase 3 performed by ModSecurity
- Perf-Phase4: %{PERF_PHASE4}M : Calculation of the performance of the rules in phase 4 performed by ModSecurity
- Perf-Phase5: %{PERF_PHASE5}M : Calculation of the performance of the rules in phase 5 performed by ModSecurity
- Perf-ReadingStorage: %{PERF_SREAD}M : The time required to read the ModSecurity session storage
- Perf-WritingStorage: %{PERF_SWRITE}M : The time required to write the ModSecurity session storage
- Perf-GarbageCollection: s%{PERF_GC}M \ The time required for garbage collection
- Perf-ModSecLogging: %{PERF_LOGGING}M : The time used by ModSecurity for logging, specifically the error log and the audit log
- Perf-ModSecCombined: %{PERF_COMBINED}M : The time ModSecurity requires in total for all work

This long list of numbers can be used to very well narrow down ModSecurity performance problems and rectify them if necessary. When you need to look even deeper, the *debug log* can help, or make use of the *PERF_RULES* variable collection, which is well explained in the reference manual.

Step 6: Writing simple blacklist rules

ModSecurity is set up and configured using the configuration above. It can diligently log performance data, but only the rudimentary basis is present on the security side. In a subsequent tutorial we will be embedding the *OWASP ModSecurity Core Rules*, a comprehensive collection of rules. But it's important for us to first learn how to write rules ourselves. Some rules have already been explained in the base configuration. It's just another small step from here.

Let's take a simple case: We want to be sure that access to a specific URI on the server is blocked. We want to respond to such a request with *HTTP status 403*. We write the rule for this in the *ModSecurity rule* section in the configuration and assign it ID 10000 (*service-specific before core-rules*).

```
SecRule  REQUEST_FILENAME "/phpmyadmin" "id:'10000',phase:1,deny,t:lowercase,t:normalisePath,\
msg:'Blocking access to %{MATCHED_VAR}.',tag:'Blacklist Rules'"
```

We start off the rule using *SecRule*. Then we say that we want to inspect the path of the request using the *REQUEST_FILENAME* variable. If */phpmyadmin* appears anywhere in this path we want to block it right away in the first processing phase. The keyword *deny* does this for us. Our path criterion is maintained in lowercase letters. Because we are using the *t:lowercase* transformation, we catch all possible lower and uppercase combinations in the path. The path could now of course also point to a subdirectory or be obfuscated in other ways. We remedy this by enabling the *t:normalisePath* transformation. The path is thus transformed before our rule is applied. We enter a message in the *msg part*, which will then show up in the server's *error log* if the rule is triggered. Finally, we assign a tag. We already did this using *SecDefaultAction* in the base configuration. There is now another tag here that can be used to group different rules.

We call this type of rules *blacklist rules*, because it describes what we want to block. In principle, we let everything pass, except for requests that violate the configured rules. The opposite approach of describing the requests we want and by doing so block all unknown requests is what we call *whitelist rules*. *Blacklist rules* are easier to write, but often remain incomplete. *Whitelist rules* are more comprehensive and when written correctly can be used to completely seal off a server. But they are difficult to write and in practice often lead to problems if they are not fully formulated. A *whitelist example* follows below.

Step 7: Trying out the blockade

Let's try out the blockade:

```
$> curl http://localhost/phpmyadmin
```

We expect the following response:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /phpmyadmin
on this server.</p>
</body></html>
```

Let's also have a look at what we can find about this in the *error log*:

```
[2015-10-27 22:43:28.265834] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 \
(phase 1). Pattern match "/phpmyadmin" at REQUEST_FILENAME. [file "/apache/conf/httpd.conf_modsec_minimal"]
[line "140"] [id "10000"] [msg "Blocking access to /phpmyadmin."] [tag "Local Lab Service"] \
[tag "Blacklist Rules"] [hostname "localhost"] [uri "/phpmyadmin"] [unique_id "Vi-wAH8AAQEAABuNHj8AAAAA"]
```

Here, *ModSecurity* describes the rule that was applied and the action taken: First the timestamp. Then the severity of the log entry assigned by Apache. The *error* stage is assigned to all *ModSecurity* messages. Then comes the client IP address. Between that there are some empty fields, indicated only by "-". In Apache 2.4 they remain empty, because the log format has changed and *ModSecurity* is not yet able to understand it. Afterwards comes the actual message which opens with action: *Access denied with code 403*, specifically already in phase 1 while receiving the request headers. We then see a message about the rule violation: The string *"/phpMyAdmin"* was found in the *REQUEST_FILENAME*. This is exactly what we defined. The subsequent bits of information are embedded in blocks of square brackets. In each block first comes the name

and then the information separated by a space. Our rule puts us on line 140 in the file */opt/apache-2.4.17/conf/httpd.conf_modsec_minimal*. As we know, the rule has ID 10000. In *msg* we see the summary of the rule defined in the rule, where the variable *MATCHED_VAR* has been replaced by the path part of the request. Afterwards comes the tag that we set in *SetDefaultAction*; finally, the tag set in addition for this rule. At the end come the hostname, URI and the unique ID of the request.

We will also find more details about this information in the *audit log* discussed above. However, for normal use the *error log* is often enough.

Step 8: Writing simple whitelist rules

Using the rules described in Step 7, we were able to prevent access to a specific URL. We will now be using the opposite approach: We want to make sure that only one specific URL can be accessed. In addition, we will we only be accepting previously known *POST parameters* in a specified format. Writing a whitelist rule such as this is done as follows:

```
SecMarker "BEGIN_LOGIN_WHITELIST"

SecRule REQUEST_FILENAME     "!@beginsWith /login" \
    "id:10001,phase:1,pass,t:lowercase,t:normalisePath,nolog,msg:'Skipping',skipAfter:END_LOGIN_WHITELIST"
SecRule REQUEST_FILENAME     "!@beginsWith /login" \
    "id:10002,phase:2,pass,t:lowercase,t:normalisePath,nolog,msg:'Skipping',skipAfter:END_LOGIN_WHITELIST"

SecRule REQUEST_FILENAME     "!^/login/(index.html|login.do)$" \
    "id:10003,phase:1,deny,log,msg:'Unknown Login URL',tag:'Whitelist Login'"

SecRule ARGS_GET_NAMES       "!^()$" \
    "id:10004,phase:1,deny,log,msg:'Unknown Query-String Parameter',tag:'Whitelist Login'"
SecRule ARGS_POST_NAMES      "!^(username|password)$" \
    "id:10005,phase:2,deny,log,msg:'Unknown Post Parameter',tag:'Whitelist Login'"

SecRule &ARGS_POST:username  "@gt 1" \
    "id:10006,phase:2,deny,log,msg:'%{MATCHED_VAR_NAME} occurring more than once',tag:'Whitelist Login'"
SecRule &ARGS_POST:password  "@gt 1" \
    "id:10007,phase:2,deny,log,msg:'%{MATCHED_VAR_NAME} occurring more than once',tag:'Whitelist Login'"

SecRule ARGS_POST:username   "!^[a-zA-Z0-9_-]{1,16}$" \
    "id:10008,phase:2,deny,log,msg:'%{MATCHED_VAR_NAME} parameter does not meet value domain'\
    ,tag:'Whitelist Login'"
SecRule ARGS_POST:password   "!^[a-zA-Z0-9@#+<>_-]{1,16}$" \
    "id:10009,phase:2,deny,log,msg:'%{MATCHED_VAR_NAME} parameter does not meet value domain'\
    ,tag:'Whitelist Login'"

SecMarker "END_LOGIN_WHITELIST"
```

Since this is a multi-line set of rules, we delimit the group of rules using two markers: *BEGIN_LOGIN_WHITELIST* and *END_LOGIN_WHITELIST*. We only need the first marker for readability, the second as a jump label. In the first rule (ID 10001) we set whether our set of rules is affected at all. If the path written in lowercase and normalized does not begin with */login*, we jump to the end marker - with no entry in the log file. This is how we circumvent the entire block of rules. (It would be possible to place the entire block of rules within an Apache *Location* block. However, I prefer the manner of writing presented here). Afterwards come our actual rules. An HTTP request has several characteristics that are of concern to us: The path, query string parameter as well as any post parameters (this concerns a login using a user name and password). We will leave out the request headers including cookies in this example, but they could also become a vulnerability depending on application and should also be queried then.

We first check the path (ID 10002). In */login* we are familiar with two paths that we accept: */login/index.html* and */login/login.do*. Everything else fails at rule 10002. Unlike the *blacklisting rules*, we now no longer need to concern ourselves with transformations. Because every path that does not match our pattern will be blocked anyway.

We then concern ourselves with the permitted query string and post parameters (IDs 10003 and 10004). We don't accept any query string parameters (the rule for this can also be simply written, but as it stands it is ready to be filled with permitted parameters). Our *username* and *password* qualify as post parameters. Any other parameter results in a blockade. Rules 10005 and 10006 handle a common method used by attackers to circumvent security rules: They send a parameter multiple times and hope that the Web Application Firewall does not check each individual occurrence and that it is used on the application server. We count the occurrence of the parameter and make sure that it does not appear more than once.

The final two rules are 10007 ad 10008. They specify patterns that the *username* and *password* parameters must match. In rule 10007 we block the message if the user name is longer than 16 characters and includes characters apart from letters, numbers, "_" and hyphens. This pattern must of course be adapted if necessary. Rule 10008 permits a few additional characters in the *password parameter*, but otherwise behaves identically.

This block of rules ensures that access to */login* is allowed only under very tight restrictions. We have thus written a basic framework of whitelisting rules that can be reused for more complicated parts of an application.

Step 9: Trying out the blockade

But does it really work? Here are some attempts:

```
$> curl http://localhost/login/index.html
-> OK (ModSecurity permits access. But this page itself does not exist. So we get 404, Page not Found)
$> curl http://localhost/login/index.html?debug=on
-> FAIL
$> curl http://localhost/login/admin.html
-> FAIL
$> curl -d "username=1234&password=test" http://localhost/login/login.do
-> OK (ModSecurity permits access. But this page itself does not exist. So we get 404, Page not Found)
$> curl -d "username=1234&password=test&backdoor=1" http://localhost/login/login.do
-> FAIL
$> curl -d "username=12345678901234567&password=test" http://localhost/login/login.do
-> FAIL
$> curl -d "username=1234'&password=test" http://localhost/login/login.do
-> FAIL
$> curl -d "username=1234&username=5678&password=test" http://localhost/login/login.do
-> FAIL
```

A glance at the server's error log proves that the are applied exactly as we defined them (excerpt filtered)

```
[2015-10-17 05:26:05.396430] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:26:07.539846] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:26:12.345245] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:26:19.976533] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:26:25.165337] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:26:42.924352] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
[2015-10-17 05:27:55.853951] [-:error] - - [client 127.0.0.1] ModSecurity: Access denied with code 403 (pha
```

It works from top to bottom.

Step 10 Goodie: Writing all client traffic to disk

Before coming to the end of this tutorial here's one more tip that often proves useful in practice: *ModSecurity* is not just a *Web Application Firewall*. It is also a very precise debugging tool. The entire traffic between client and server can be logged. This is done as follows:

```
SecRule REMOTE_ADDR  "@streq 127.0.0.1"   "id:11000,phase:1,pass,log,auditlog,\
    msg:'Initializing full traffic log'"
```

We then find the traffic for the client 127.0.0.1 specified in the rule in the audit log.

```
$> curl localhost
...
$> sudo tail -1 /apache/logs/modsec_audit.log
localhost 127.0.0.1 - - [17/Oct/2015:06:17:08 +0200] "GET /index.html HTTP/1.1" 404 214 "-" "-" UcAmDH8AAQE
$> sudo cat /apache/logs/audit/20151017/20151017-0617/20151017-061708-UcAmDH8AAQEAAGUjAMoAAAAA
--c54d6c5e-A--
[17/Oct/2015:06:17:08 +0200] UcAmDH8AAQEAAGUjAMoAAAAA 127.0.0.1 52386 127.0.0.1 80
--c54d6c5e-B--
GET /index.html HTTP/1.1
User-Agent: curl/7.35.0 (x86_64-pc-linux-gnu) libcurl/7.35.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp
Host: localhost
Accept: */*
```

```
--c54d6c5e-F--
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2015 21:39:03 GMT
Server: Apache
Last-Modified: Tue, 06 Oct 2015 11:55:08 GMT
ETag: "2d-5216e4d2e6c03"
Accept-Ranges: bytes
Content-Length: 45

--c54d6c5e-E--
<html><body><h1>It works!</h1></body></html>
...
```

The rule that logs traffic can of course be customized, enabling us to precisely see what goes into the server and what it returns (only a specific client IP, a specific user, only a application part with a specific path, etc.). It often allows you to quickly find out about the misbehavior of an application.

We have reached the end of this tutorial. *ModSecurity* is an important component for the operation of a secure web server. This tutorial has hopefully provided a successful introduction to the topic.

References

- Apache https://httpd.apache.org
- ModSecurity https://www.modsecurity.org
- ModSecurity Reference Manual https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual

License / Copying / Further use