Tutorial 7 - Embedding OWASP ModSecurity Core Rules

What are we doing?

We are embedding the OWASP ModSecurity Core Rules in our Apache web server and eliminating false alarms.

Why are we doing this?

The ModSecurity Web Application Firewall, as we set up in Tutorial 6, still has barely any rules. But protection only works when you configure an additional rule set that is as comprehensive as possible and when you have eliminated all of the false alarms. The core rules provide generic blacklisting. This means that they inspect requests and responses for signs of attacks. The signs are often keywords or typical patterns that may be suggestive of a wide variety of attacks. This also entails false alarms being triggered.

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 Apache web server with extended access log as in Tutorial 5 (Extending and analyzing the access log)
- An Apache web server with ModSecurity as in Tutorial 6 (Embedding ModSecurity)

Step 1: Downloading OWASP ModSecurity Core Rules

The ModSecurity Core Rules are being developed under the umbrella of *OWASP*, the Open Web Application Security Project. The rules themselves are available at *GitHub* and can be downloaded as follows.

```
$> wget https://github.com/SpiderLabs/owasp-modsecurity-crs/archive/2.2.9.tar.gz
$> tar xvzf 2.2.9.tar.gz
owasp-modsecurity-crs-2.2.9/
owasp-modsecurity-crs-2.2.9/.gitignore
owasp-modsecurity-crs-2.2.9/CHANGES
owasp-modsecurity-crs-2.2.9/INSTALL
owasp-modsecurity-crs-2.2.9/LICENSE
owasp-modsecurity-crs-2.2.9/README.md
...
$> sudo mkdir /opt/modsecurity-core-rules-2.2.9
$> sudo chown `whoami` /opt/modsecurity-core-rules-2.2.9
$> cp owasp-modsecurity-crs-2.2.9/base_rules/* /opt/modsecurity-core-rules-2.2.9
$> sudo ln -s /opt/modsecurity-core-rules-2.2.9 /modsecurity-core-rules
$> rm -r 2.2.9.tar.gz owasp-modsecurity-crs-2.2.9
```

This unpacks the base part of the core rules in the directory `/opt/modsecurity-core-rules-2.2.9`. We create a link from `/modsecurity-core-rules` to this directory to do this. We then delete the rest of the core rules package. There are in fact a wide variety of optional rules which may be of interest depending on the situation. We will however ignore them while getting started in our lab-like setup. Furthermore, we will skip over a file named `modsecurity_crs_10_setup.conf`. A base configuration is usually written in this file and the file is then imported by Apache via *Include*. But this has not been a successful approach in my opinion. In the next section we will be directly integrating the content of this file in our Apache configuration, so we won't be needing the file itself.

Step 2: Embedding Core Rules

In Tutorial 6, in which we embedded ModSecurity itself, we marked out a section for the core rules. We now add the Include directive in this section. Specifically, four parts are added to the existing configuration. (1) The core rules base configuration, (2) a part for self-defined ignore rules before the core rules. Then (3) the core rules themselves and finally a part (4) for self-defined ignore rules after the core rules.

The ignore rules are rules used for managing the false alarms described above. Some false alarms must be prevented before the corresponding core rule is loaded. Some false alarms can only be intercepted following the definition of the core rule itself. But one thing at a time. First off, here's the complete configuration file:

```
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-WritingStorage: %{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_perflog"



# === 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 (ids: 900000-999999)

# === ModSec Core Rules Base Configuration (ids: 900001-900021)

SecAction "id:'900001',phase:1,t:none, \
   setvar:tx.critical_anomaly_score=5, \
   setvar:tx.error_anomaly_score=4, \
   setvar:tx.warning_anomaly_score=3, \
   setvar:tx.notice_anomaly_score=2, \
   nolog, pass"
SecAction "id:'900002',phase:1,t:none,setvar:tx.inbound_anomaly_score_level=10000,\
    setvar:tx.inbound_anomaly_score=0,nolog,pass"
SecAction "id:'900003',phase:1,t:none,setvar:tx.outbound_anomaly_score_level=10000,\
    setvar:tx.outbound_anomaly_score=0,nolog,pass"
SecAction "id:'900004',phase:1,t:none,setvar:tx.anomaly_score_blocking=on,nolog,pass"

SecAction "id:'900006',phase:1,t:none,setvar:tx.max_num_args=255,nolog,pass"
SecAction "id:'900007',phase:1,t:none,setvar:tx.arg_name_length=100,nolog,pass"
SecAction "id:'900008',phase:1,t:none,setvar:tx.arg_length=400,nolog,pass"
SecAction "id:'900009',phase:1,t:none,setvar:tx.total_arg_length=64000,nolog,pass"
SecAction "id:'900010',phase:1,t:none,setvar:tx.max_file_size=10000000,nolog,pass"
SecAction "id:'900011',phase:1,t:none,setvar:tx.combined_file_sizes=10000000,nolog,pass"
SecAction "id:'900012',phase:1,t:none, \
  setvar:'tx.allowed_methods=GET HEAD POST OPTIONS', \
  setvar:'tx.allowed_request_content_type=application/x-www-form-urlencoded|\
multipart/form-data|text/xml|application/xml|application/x-amf|application/json', \
  setvar:'tx.allowed_http_versions=HTTP/0.9 HTTP/1.0 HTTP/1.1', \
  setvar:'tx.restricted_extensions=.asa/ .asax/ .ascx/ .axd/ .backup/ .bak/ .bat/ \
.cdx/ .cer/ .cfg/ .cmd/ .com/ .config/ .conf/ .cs/ .csproj/ .csr/ .dat/ .db/ .dbf/ \
.dll/ .dos/ .htr/ .htw/ .ida/ .idc/ .idq/ .inc/ .ini/ .key/ .licx/ .lnk/ .log/ .mdb/ \
.old/ .pass/ .pdb/ .pol/ .printer/ .pwd/ .resources/ .resx/ .sql/ .sys/ .vb/ .vbs/ \
.vbproj/ .vsdisco/ .webinfo/ .xsd/ .xsx/', \
  setvar:'tx.restricted_headers=/Proxy-Connection/ /Lock-Token/ /Content-Range/ \
/Translate/ /via/ /if/', \
  nolog,pass"

SecRule REQUEST_HEADERS:User-Agent "^(.*)$" "id:'900018',phase:1,t:none,t:sha1,t:hexEncode,\
setvar:tx.ua_hash=%{matched_var}, \
  nolog,pass"
SecRule REQUEST_HEADERS:x-forwarded-for "^\b(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b" \
  "id:'900019',phase:1,t:none,capture,setvar:tx.real_ip=%{tx.1},nolog,pass"
SecRule &TX:REAL_IP "!@eq 0" "id:'900020',phase:1,t:none,initcol:global=global,\
initcol:ip=%{tx.real_ip}_%{tx.ua_hash}, \
  nolog,pass"
SecRule &TX:REAL_IP "@eq 0" "id:'900021',phase:1,t:none,initcol:global=global,\
initcol:ip=%{remote_addr}_%{tx.ua_hash},setvar:tx.real_ip=%{remote_addr}, \
  nolog,pass"

# === ModSecurity Ignore Rules Before Core Rules Inclusion; order by id of ignored rule (ids: 10000-49999)

# ...

# === ModSecurity Core Rules Inclusion

Include    /modsecurity-core-rules/*.conf

# === ModSecurity Ignore Rules After Core Rules Inclusion; order by id of ignored rule (ids: 50000-59999)

# ...

# === 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>
```

In the base configuration we define a variety of values which are queried and used by the core rules. In rule *900001* the different degrees of anomaly are assigned numerical values called scores. A "critical error" is given a 5, an error at the *error* level gets a 4, 3 for a "warning" and a notice gets a score of 2. An HTTP request passes through the core rules like a large filter. Each individual core rule is assigned an anomaly score. If, for instance, a request violates a *critical* rule, the request is given a score of 5. One request can violate multiple rules and the same rule can be repeatedly violated when different parameters are queried and the rule kicks in multiple times. The *anomaly scores* are then summed up for each request and separately for each request and response. Quite a large total can be reached on untuned systems; scores of over 500 are no

rarity and even over 1000 have been seen.

In rules *900002* and *900003* we define the limits at which a request should be blocked. One limit is for the requests (*inbound*), a second limit for the responses (*outbound*). To start off, we define a very high value of 10000 for both limits. In practice this means that the limits will never be reached. In rule *900004* we formally enable blocking mode. This is where we could also define that we don't want to block anything at all, but want to work in monitoring mode. However, I strongly urge against this. We should be working in blocking mode from the very beginning and reduce the limits step-by-step. Reducing the limits in monitoring mode frequently gets stuck half-way and if you succeed in reducing them nevertheless, in the end you often won't feel confident in switching to blocking mode. It is thus better to start off in blocking mode and to build up more confidence in the installation at every limit reduction.

In summary, we have now chosen a blocking mode with very lax limits for the core rules. We can tighten up the limits at a later point in time; but as far as the blocking principle goes, there's nothing more we need to change.

In rules *900006* to *900009* we set, in order, the maximum number of request parameters, the character length of parameter names, the length of a parameter and finally, the combined length of all parameters. The last value roughly approximates the values set for *SecRequestBodyLimit* and *SecRequestBodyNoFilesLimit* in the file above. These two values represent hard limits, while parameters *tx.max_file_size* and *tx.combined_file_sizes=1000* are a bit softer, by simply violating a *core rule* of severity *notice* (a score of 2). This is a server-side warning in the log and not a blockade. If you want to be tough, then *SecRequestBodyLimit* and *SecRequestBodyNoFilesLimit* are the values for you.

This also applies to rules *900010* and *900011*, which define maximum and combined file sizes. The permitted HTTP methods are listed in rule *900012*, because we are no longer accepting all methods. Then come the media types allowed in requests: These are primarily the standard *application/x-www-form-urlencoded* and *multipart/form-data* used for file uploads. Added to this are two or three variations of XML and *application/json*. After that come the acceptable HTTP versions, then a list of unwanted file extensions and finally, restricted request headers.

Now come rules *900018* to and including *900021*. These rules are very demanding. The work together and create what are called collections. These are collections of data retained beyond a single request. They are used to capture and monitor user sessions. In rule *900018* the *User-Agent header* from the request is converted to a hash using the SHA-1 method and then encoded into hexadecimal form. This value, called *ua_hash* in this case, or the user-agent hash, is written to a variable. This is done via *%{matched_var}*, an internal *ModSecurity* variable that represents the value in the conditional part of the rule. *%{matched_var}* is always included in a *SecRule* directive. Afterwards comes rule *900019* which watches out for *X-Forwarded-For* headers. *X-Forward-For* headers are written by proxies, which typically disallow HTTP clients in corporate networks. The original IP address of the client is then included in the header itself. If present, we take the IP address and set the variable *real_ip* accordingly. But unlike in the previous rule, we no longer use *%{matched_var}*, but select the value present in the brackets of the regular expression. We use the action *capture* and then access the first bracket using *%{tx.1}*. *TX* is the abbreviation for *transaction* here. What is meant is a collection related to the current request and the current matches of the regular expression. If this works, then in *90020* we use the value *global* to initialize the *GLOBAL* collection and at the same time initialize the collection IP using the key composed from *real_ip* and *ua_hash*. If no *X-Forward-For* header is present, then in rule *900021* we set the collection *IP* to the IP address of the TCP connection, the values of the internal *REMOTE_ADDR* variable, or *remote_addr* in this case.

All required variables are now initialized and we are ready to load the *OWASP ModSecurity Core Rules*, which we have already prepared. However, the block for the future handling of false alarms previously mentioned comes before the required *include directive* in the configuration file. This block currently consists of a comment and a placeholder. Then comes the actual *include directive* for the core rules, in turn followed by a comment as a placeholder for the future handling of false alarms. False alarms can be fought against in a number of different ways. This sometimes has to happen before loading the core rules and sometimes after they have already been loaded. That's why we are setting up two places for these directives.

We have embedded the core rules and are now ready for test operation. The rules inspect requests and responses. They also trigger alarms, but will not block any requests, because the limits have been set very high. It can be very easy to distinguish between the triggering of alarms and actual blocking in the Apache error log. Particularly since the individual core rules, as we have seen, increase by one anomaly score, but do not trigger a blockade. The blockade itself is performed by a separate blocking rule taking the limits into account. This won't work for the moment, however. ModSecurity logs normal rule violations in the error log as *ModSecurity. Warning ...*, and blockades as *ModSecurity. Access denied ...*. As long as no *Access denied* is logged, users are working without being impacted by ModSecurity.

Step 3: Triggering alarms for testing purposes

We have now equipped our web server with a complete WAF installation. What's still left to do is to fine tune, or tweak, the

configuration: To get rid of false alarms. But we should first see what alarms really look like. Let's run a simple vulnerability scanner on our test installation. *Nikto* is just such a simple tool that can quickly give us results. *Nikto* may still have to be installed. The scanner is however included in most distributions.

```
$> nikto -h localhost
- Nikto v2.1.4
---------------------------------------------------------------------------
+ Target IP:          127.0.0.1
+ Target Hostname:    localhost
+ Target Port:        80
+ Start Time:         2015-11-08 08:33:59
---------------------------------------------------------------------------
+ Server: Apache
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ ETag header found on server, fields: 0x2d 0x432a5e4a73a80
+ Allowed HTTP Methods: POST, OPTIONS, GET, HEAD
+ 6448 items checked: 0 error(s) and 2 item(s) reported on remote host
+ End Time:           2015-11-08 08:34:14 (15 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

This scan should have triggered numerous *ModSecurity alarms* on the server. Let's take a close look at the *Apache error log*. In my case there were over 33,000 entries in the error log. An example message:

```
[2015-11-07 08:34:13.816811] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Pattern match "(fromcha
```

The *error log* was discussed in the preceding tutorial. The messages triggered by the core rules include more information than default messages, making it perhaps worthwhile to discuss the log format once more.

The beginning of the line consists of the Apache-specific parts such as the timestamp and the severity of the message as the Apache server sees it. Unfortunately, some fields are still empty (-> "-"). This is due to a bug in *ModSecurity* that should be fixed in the next version. *ModSecurity* messages are always set to *error* level. Then comes the client IP address. Afterwards a reference to ModSecurity. It's important to know here that the message *ModSecurity: Warning* is really just a warning. When the module intervenes in traffic, it then writes *ModSecurity: Access denied with code ...*. This is a differentiation you can rely on. A *warning* never has any direct impact on the client.

What comes next? A reference to the pattern found in the request. A specific pattern for a regular expression was found in request argument *ctr* . Then comes a series of parameters that always have the same pattern: They are within square brackets and have their own identifier. First comes the *file* identifier. It shows us the file in which the rule that triggered the alarm is defined. This is followed by *line* for the line number in the file. The *id* parameter appears more important to me. Every rule in *ModSecurity* has an identification number and can thus be uniquely identified. Then comes *rev* as a reference to the revision number of the rule. In core rules these parameters express how often the rule has been revised. If a modification is made to a rule, *rev* increases by one. *msg*, short for *message*, describes the type of attack detected. The relevant part of the request, the *ctr* parameter appears in *data*. In my example this is obviously a case of *cross site scripting* (XSS).

At *ver* we come to the release of the core rule set, followed by *maturity*, a reference to the quality of the rule. Higher *maturity* indicates that we can trust this rule, because it is in widespread use and has caused very few problems. However, low *maturity* is likely to indicate an experimental rule. This is why the value 1 appears only six times in the *core rules*, whereas in the version being used 116 rules have a value of 8 and 99 rules are assumed to have a maturity of 9. *Accuracy*, the precision of the rule, behaves similar to *maturity*. This is an optional value the author of the rule defined when writing the rule. There are no low values in the system of rules, 8 is the most frequent value (144 times), 9 is widespread (82). These additional notes in the log message are for documentation purposes only. In my experience they are of little relevance and hardly ever change between *Core Rules* releases.

Now comes a series of *tags* assigned to the rule. First comes the *Local Lab Service* tag. We defined this one ourselves in our configuration. It is therefore being included along with every rule violation. Afterwards follow several tags from the *Core Rule Set*, classifying the type of attack. These references can, for example, be used for analysis and statistics.

Towards the end of the alarm come three additional values, *hostname*, *uri* and *unique_id*, that more clearly specify the request. We can use *unique_id* to make the connection to our *access log* and *URI* helps us to find the resource on our server.

A single alarm includes a great deal of information. Over 30,000 entries for a single invocation of *nikto* adds up to 23 MB of data. You are well-advised to keep an eye on the size of the *error log*.

Step 4: Analyzing anomaly scores

Although the format of the entries in the error log may be clear, without a tool they are very hard to read. A simple remedy is to use a few *shell aliases*, which extract individual pieces of information from the entries. They are stored in the alias file we discussed in Tutorial 5.

```
$> cat ~/.apache-modsec.alias
...
alias meldata='grep -o "\[data [^]]*" | cut -d\" -f2'
alias melfile='grep -o "\[file [^]]*" | cut -d\" -f2'
alias melhostname='grep -o "\[hostname [^]]*" | cut -d\" -f2'
alias melid='grep -o "\[id [^]]*" | cut -d\" -f2'
alias melip='grep -o "\[client [^]]*" | cut -b9-'
alias melline='grep -o "\[line [^]]*" | cut -d\" -f2'
alias melmatch='grep -o " at [^\ ]*\. \[file" | sed -e "s/\. \[file//" | cut -b5-'
alias melmsg='grep -o "\[msg [^]]*" | cut -d\" -f2'
alias meltimestamp='cut -b2-25'
alias melunique_id='grep -o "\[unique_id [^]]*" | cut -d\" -f2'
alias meluri='grep -o "\[uri [^]]*" | cut -d\" -f2'
...
$> source ~/.apache-modsec.alias
```

These abbreviations all start with the prefix *mel*, short for *ModSecurity error log*. Then comes the field name. Let's try it out to output the rule IDs in the messages.

```
$> cat logs/error.log | melid | head
960015
990002
990012
960024
950005
981173
981243
981203
960901
960015
990002
```

This seems to do the job. So let's extend the example in a few steps:

```
$> cat logs/error.log | melid | sort | uniq -c | sort -n
      1 950000
      1 950107
      1 950907
      1 950921
      1 960007
      1 960208
      1 960209
      1 981202
      1 981319
      2 958031
      2 959071
      2 960008
      2 973304
      2 973334
      3 950011
      3 973338
      3 981240
      3 981260
      3 981276
      3 981317
      5 950001
      5 959073
      5 960010
      6 950006
      6 981257
      7 970901
      8 981249
      9 981242
```

```
     11 960032
     13 960034
     15 973305
     15 973346
     17 960011
     17 960911
     19 981227
     29 981246
     64 973335
     67 950109
     68 960901
     75 981231
     77 981245
    106 958001
    141 950118
    155 981243
    162 981318
    179 950103
    219 960035
    225 950005
    231 973336
    245 958051
    245 973331
    247 950901
    248 973300
    284 958052
    284 973307
    418 960024
    531 981173
   2274 950119
   2335 950120
   6133 960015
   6134 981203
   6135 990002
   6135 990012
$> cat logs/error.log | melid | sort | uniq -c | sort -n  | while read STR; do echo -n "$STR "; \
ID=$(echo "$STR" | sed -e "s/.*\ //"); grep $ID logs/error.log | head -1 | melmsg; done
1 950000 Session Fixation
1 950107 URL Encoding Abuse Attack Attempt
1 950907 System Command Injection
1 950921 Backdoor access
1 960007 Empty Host Header
1 960208 Argument value too long
1 960209 Argument name too long
1 981202 Correlated Attack Attempt Identified: (Total Score: 22, SQLi=5, XSS=) Inbound Attack ...
1 981319 SQL Injection Attack: SQL Operator Detected
2 958031 Cross-site Scripting (XSS) Attack
2 959071 SQL Injection Attack
2 960008 Request Missing a Host Header
2 973304 XSS Attack Detected
2 973334 IE XSS Filters - Attack Detected.
3 950011 SSI injection Attack
3 973338 XSS Filter - Category 3: Javascript URI Vector
3 981240 Detects MySQL comments, conditions and ch(a)r injections
3 981260 SQL Hex Encoding Identified
3 981276 Looking for basic sql injection. Common attack string for mysql, oracle and others.
3 981317 SQL SELECT Statement Anomaly Detection Alert
5 950001 SQL Injection Attack
5 959073 SQL Injection Attack
5 960010 Request content type is not allowed by policy
6 950006 System Command Injection
6 981257 Detects MySQL comment-/space-obfuscated injections and backtick termination
7 970901 The application is not available
8 981249 Detects chained SQL injection attempts 2/2
9 981242 Detects classic SQL injection probings 1/2
11 960032 Method is not allowed by policy
13 960034 HTTP protocol version is not allowed by policy
15 973305 XSS Attack Detected
15 973346 IE XSS Filters - Attack Detected.
17 960011 GET or HEAD Request with Body Content.
17 960911 Invalid HTTP Request Line
19 981227 Apache Error: Invalid URI in Request.
29 981246 Detects basic SQL authentication bypass attempts 3/3
64 973335 IE XSS Filters - Attack Detected.
```

```
67 950109 Multiple URL Encoding Detected
68 960901 Invalid character in request
75 981231 SQL Comment Sequence Detected.
77 981245 Detects basic SQL authentication bypass attempts 2/3
106 958001 Cross-site Scripting (XSS) Attack
141 950118 Remote File Inclusion Attack
155 981243 Detects classic SQL injection probings 2/2
162 981318 SQL Injection Attack: Common Injection Testing Detected
179 950103 Path Traversal Attack
219 960035 URL file extension is restricted by policy
225 950005 Remote File Access Attempt
231 973336 XSS Filter - Category 1: Script Tag Vector
245 958051 Cross-site Scripting (XSS) Attack
245 973331 IE XSS Filters - Attack Detected.
247 950901 SQL Injection Attack: SQL Tautology Detected.
248 973300 Possible XSS Attack Detected - HTML Tag Handler
284 958052 Cross-site Scripting (XSS) Attack
284 973307 XSS Attack Detected
418 960024 Meta-Character Anomaly Detection Alert - Repetitive Non-Word Characters
531 981173 Restricted SQL Character Anomaly Detection Alert - Total # of special characters exceeded
2274 950119 Remote File Inclusion Attack
2335 950120 Possible Remote File Inclusion (RFI) Attack: Off-Domain Reference/Link
6133 960015 Request Missing an Accept Header
6134 981203 Inbound Anomaly Score (Total Inbound Score: 10, SQLi=, XSS=): Rogue web site crawler
6135 990002 Request Indicates a Security Scanner Scanned the Site
6135 990012 Rogue web site crawler
```

This we can work with. But it's perhaps necessary to explain the *one-liners*. We extract the rule IDs from the *error log*, then *sort* them, put them together in a list of found IDs (*uniq -c*) and sort again by the numbers found. That's the first *one-liner*. A relationship between the individual rules is still lacking, because there's not much we can do with the ID number yet. We get the names from the *error log* again by looking through the previously run test line-by-line in a loop. We show what we have in this loop. We then have to separate the number of found items and the IDs again. This is done using an embedded sub-command (`ID=$(echo "$STR" | sed -e "s/.*\ //")`). We then use the IDs we just found to search the *error log* once more for an entry, but take only the first one, extract the *msg* part and display it. Done.

Depending on computing power this may take just a few seconds or a few minutes. You might now think that it would be better to define an additional alias to determine the ID and description of the rule in a single step. This puts us on the wrong path, because rule 981203 has the identifier containing dynamic parts in and following the brackets. We of course want to put them together them in order to map the rule only once. So, to really simplify analysis we have to get rid of the dynamic items. Here's an additional *alias* that implements this idea. It is part of the *.apache-modsec.alias* file you are already familiar with.

```
alias melidmsg='grep -o "\[id [^]]*\].*\[msg [^]]*\]" | sed -e "s/\].*\[/] [/" | cut -b6-11,19- | \
tr -d \] | sed -e "s/(Total .*/(Total ...) .../" | tr -d \"'
```

```
$> cat logs/error.log | melidmsg | sucs
      1 950000 Session Fixation
      1 950107 URL Encoding Abuse Attack Attempt
      1 950907 System Command Injection
      1 950921 Backdoor access
      1 960007 Empty Host Header
      1 960208 Argument value too long
      1 960209 Argument name too long
      1 981202 Correlated Attack Attempt Identified: (Total ...) ...
      1 981319 SQL Injection Attack: SQL Operator Detected
      2 958031 Cross-site Scripting (XSS) Attack
      2 959071 SQL Injection Attack
      2 960008 Request Missing a Host Header
      2 973304 XSS Attack Detected
      2 973334 IE XSS Filters - Attack Detected.
      3 950011 SSI injection Attack
      3 973338 XSS Filter - Category 3: Javascript URI Vector
      3 981240 Detects MySQL comments, conditions and ch(a)r injections
      3 981260 SQL Hex Encoding Identified
      3 981276 Looking for basic sql injection. Common attack string for mysql, oracle and others.
      3 981317 SQL SELECT Statement Anomaly Detection Alert
      5 950001 SQL Injection Attack
      5 959073 SQL Injection Attack
      5 960010 Request content type is not allowed by policy
```

```
   6 950006 System Command Injection
   6 981257 Detects MySQL comment-/space-obfuscated injections and backtick termination
   7 970901 The application is not available
   8 981249 Detects chained SQL injection attempts 2/2
   9 981242 Detects classic SQL injection probings 1/2
  11 960032 Method is not allowed by policy
  13 960034 HTTP protocol version is not allowed by policy
  15 973305 XSS Attack Detected
  15 973346 IE XSS Filters - Attack Detected.
  17 960011 GET or HEAD Request with Body Content.
  17 960911 Invalid HTTP Request Line
  19 981227 Apache Error: Invalid URI in Request.
  29 981246 Detects basic SQL authentication bypass attempts 3/3
  64 973335 IE XSS Filters - Attack Detected.
  67 950109 Multiple URL Encoding Detected
  68 960901 Invalid character in request
  75 981231 SQL Comment Sequence Detected.
  77 981245 Detects basic SQL authentication bypass attempts 2/3
 106 958001 Cross-site Scripting (XSS) Attack
 141 950118 Remote File Inclusion Attack
 155 981243 Detects classic SQL injection probings 2/2
 162 981318 SQL Injection Attack: Common Injection Testing Detected
 179 950103 Path Traversal Attack
 219 960035 URL file extension is restricted by policy
 225 950005 Remote File Access Attempt
 231 973336 XSS Filter - Category 1: Script Tag Vector
 245 958051 Cross-site Scripting (XSS) Attack
 245 973331 IE XSS Filters - Attack Detected.
 247 950901 SQL Injection Attack: SQL Tautology Detected.
 248 973300 Possible XSS Attack Detected - HTML Tag Handler
 284 958052 Cross-site Scripting (XSS) Attack
 284 973307 XSS Attack Detected
 418 960024 Meta-Character Anomaly Detection Alert - Repetitive Non-Word Characters
 531 981173 Restricted SQL Character Anomaly Detection Alert - Total # of special characters exceeded
2274 950119 Remote File Inclusion Attack
2335 950120 Possible Remote File Inclusion (RFI) Attack: Off-Domain Reference/Link
6133 960015 Request Missing an Accept Header
6134 981203 Inbound Anomaly Score (Total ...) ...
6135 990002 Request Indicates a Security Scanner Scanned the Site
6135 990012 Rogue web site crawler
```

Step 5: Evaluating false alarms

Our *Nikto* scan set off thousands of alarms. They were likely justified. In the normal use of *ModSecurity* things are different: Depending on application, a normal installation will also see a lot of alarms and in my experience most of them are false. The configuration has to first be fine tuned to ensure smooth operation. We want to achieve a high degree of separation. We wish to configure *ModSecurity* so the engine knows exactly how to distinguish between legitimate requests and attacks.

False alarms are possible in both directions. Attacks that are not detected are called *false negatives*. The *core rules* are strict and careful to keep the number of *false negatives* low. An attacker would have to possess a great deal of savvy to circumvent the system of rules. Unfortunately, this strictness also results in alarms being triggered for normal requests. There are a lot of *false positives* like these. It is commonly the case that at a low degree of separation you either get a lot of *false negatives* or a lot of *false positives*. Reducing the number of *false negatives* leads to an increase in *false positives*. Both correlate highly with one another.

We have to overcome this link: We want to increase the degree of separation in order to reduce the number of *false positives* without increasing the number of *false negatives*. We can do this by fine tuning the system of rules in a few places. But first we need to have a clear picture of the current situation: How many *false positives* are there and which of the rules are being violated in a particular context? We need a plan or a target. How many *false positives* are we willing to allow on the system? Reducing them to zero will be extremely difficult to do, but percentages are something we can work with. A possible target would be: 99.99% of legitimate requests should pass without being blocked. This is realistic, but involves a bit of work depending on the application.

To reach such a target we will need one or two tools to help us get a good footing. Specifically, we want to find out the *anomaly scores* the different requests to the server have been assigned and which of the rules have actually been violated. We have modified the *log format* in such a way that it is easy for the *anomaly scores* to be extracted from the *access log*. We now want to present these data in a suitable form.

In Tutorial 4 we worked with a sample log file containing 10,000 entries. We'll be using this log file again here: tutorial-5-example-access.log. The file comes from a real server, but the IP addresses, server names and paths have been simplified or rewritten. However, the information we need for our analysis is still there. Let's have a look at the distribution of *anomaly scores*:

```
$> egrep -o "[0-9-]+ [0-9-]+$" tutorial-5-example-access.log | cut -d\  -f1 | sucs
      1 21
      2 41
      8 5
     11 2
     17 3
     41 -
   9920 0
$> egrep -o "[0-9-]+$" tutorial-5-example-access.log | sucs
     41 -
   9959 0
```

The first command line reads the inbound *anomaly score*. It's the second-to-last value in the *access log line*. We take the two last values (*egrep*) and then *cut* the first one out. We then sort the results using the familiar *sucs* alias. The outbound *anomaly score* is the last value in the *log line*. This is why there is no *cut* command on the second command line.

The results give us an idea about the situation: The great majority of requests pass the *ModSecurity module* with no rule violation. A score of 41 appears twice, corresponding to a high number of serious rule infractions. This is very common in practice. In 41 cases we didn't get any score for the server's responses. These are log entries of empty requests in which a connection with the client was established, but no request was made. We have taken this possibility into account in the regular expression using *egrep* and are also taking account of the default value "-". Besides these empty entries, nothing else is conspicuous at all. This is typical. In all likelihood we will be seeing a high number of violations from the requests and very few alarms from the responses.

But this still doesn't give us the right idea about the *tuning steps* needed. To present this information in suitable form I have prepared a script that analyzes *anomaly scores*. modsec-positive-stats.rb. Applied to the log file, the script delivers the following result:

```
$> cat tutorial-5-example-access.log  | egrep -o "[0-9-]+ [0-9-]+$" | tr " " ";" | modsec-positive-stats.rb
INCOMING                     Num of req. | % of req. |  Sum of % | Missing %
Number of incoming req. (total) |  10000 | 100.0000% | 100.0000% |   0.0000%

Empty or miss. incoming score   |     41 |   0.4100% |   0.4100% |  99.5900%
Reqs with incoming score of   0 |   9920 |  99.2000% |  99.6100% |   0.3900%
Reqs with incoming score of   1 |      0 |   0.0000% |  99.6100% |   0.3900%
Reqs with incoming score of   2 |     11 |   0.1100% |  99.7200% |   0.2800%
Reqs with incoming score of   3 |     17 |   0.1699% |  99.8900% |   0.1100%
Reqs with incoming score of   4 |      0 |   0.0000% |  99.8900% |   0.1100%
Reqs with incoming score of   5 |      8 |   0.0800% |  99.9700% |   0.0300%
Reqs with incoming score of   6 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of   7 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of   8 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of   9 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  10 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  11 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  12 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  13 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  14 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  15 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  16 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  17 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  18 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  19 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  20 |      0 |   0.0000% |  99.9700% |   0.0300%
Reqs with incoming score of  21 |      1 |   0.0100% |  99.9800% |   0.0200%
Reqs with incoming score of  22 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  23 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  24 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  25 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  26 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  27 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  28 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  29 |      0 |   0.0000% |  99.9800% |   0.0200%
```

```
Reqs with incoming score of  30 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  31 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  32 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  33 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  34 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  35 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  36 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  37 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  38 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  39 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  40 |      0 |   0.0000% |  99.9800% |   0.0200%
Reqs with incoming score of  41 |      2 |   0.0200% | 100.0000% |   0.0000%

Average:   0.0217        Median   0.0000         Standard deviation   0.6490


OUTGOING                     Num of req. | % of req. |  Sum of % | Missing %
Number of outgoing req. (total) |  10000 | 100.0000% | 100.0000% |   0.0000%

Empty or miss. incoming score   |      41|  0.4100%  |   0.4100% |  99.5900%
Reqs with outgoing score of   0 |   9959 |  99.5900% | 100.0000% |   0.0000%

Average:   0.0000        Median   0.0000         Standard deviation   0.0000
```

The script divides the incoming from the outgoing *anomaly scores*. The incoming ones are handled first. At first, one line describes how often an empty *anomaly score* has been found (*empty incoming score*). In our case this was the 41 times we saw before. Then comes the statement about *score 0*: 9920 requests. This is coverage of 99.2%. Together with the empty scores this is already coverage of 99.61% (*Sum of %*). 0.39% had a higher *anomaly score* (*Missing %*). Above we set out to have 99.99% of requests able to pass the server. We are just 0.38% or 38 requests away from this target. The next *anomaly score* is 2. It appears 11 times or 0.11%. The 3 appears 17 times and a score of 5 8 times. All in all, we are at 99.97%. Then there is one request with a score of 21 and finally 2 requests with 41. To achieve 99.99% coverage we have get to this limit (and, based on the log file, thus achieve 100% coverage).

There are probably some *false positives*. In practice, we have to make certain of this before we start fine tuning the rules. It would basically be wrong to assume a false positive based on a justified alarm and suppress the alarm in the future. Before tuning it must therefore be ensured that no attacks are present in the log file. This is not always very easy. Manual review helps, restricting to known IP addresses, testing/tuning on a test system separated from the internet, filtering the access log by country of origin for the IP address, etc.: It's a big topic and making general recommendations is difficult.

Step 6: Suppressing false alarms: Disabling individual rules

This simple way of dealing with a *false positive* is to simply disable the rule. This takes very little effort, but is of course potentially risky, because the rule is not being disabled for just legitimate users, but for attackers as well. By completely disabling a rule we are thus restricting the capability of *ModSecurity*. Or, expressed more drastically, we're pulling the teeth out of the *WAF*. This is not something we want in practice. Nevertheless, it is important to know how this simple method works.

Above we saw a list of alarms we can use to provoke the *Nikto* security scanner. One rule, which *Nikto* along with legitimate browsers sometimes violate is 960015: Request Missing an Accept Header. Due to this rule, alarms on a normal server occur quite frequently. This is a reason for disabling the rule.

In our configuration file we have marked out two places to put the *ignore rules*. Once before the *core rules* and a second time after the *core rules*:

```
# === ModSecurity Ignore Rules Before Core Rules Inclusion; order by id of ignored rule (ids: 10000-49999)

...

# === ModSecurity Core Rules Inclusion

Include    conf/modsecurity-core-rules-latest/*.conf

# === ModSecurity Ignore Rules After Core Rules Inclusion; order by id of ignored rule (ids: 50000-79999)

...
```

We are suppressing rule *960015* in the section above. Before we do this, we provoke an alarm for the rule:

```
$> curl -v -H "Accept: " http://localhost/index.html
...
> GET /index.html HTTP/1.1
> User-Agent: curl/7.32.0
> Host: localhost
...
$> tail /apache/logs/error.log
...
[Tue Dec 10 06:41:41 2013] [error] [client 127.0.0.1] ModSecurity: Warning. Operator EQ matched 0 at REQUES
[Tue Dec 10 06:41:41 2013] [error] [client 127.0.0.1] ModSecurity: Warning. Operator LT matched 1000 at TX:
```

We instructed *curl* to send a request with no *Accept header*. Using the *verbose* option (*-v*) we can perfectly control this behavior. The *error log* will then actually show the alarm that was provoked with the summary of *anomaly scores* on the lines below. The rule violation earned the request a score of 2. Now we'll be suppressing the rule by writing an *ignore rule* in the configuration section provided for it before *Core Rules Inclusion*:

```
SecRule REQUEST_FILENAME "@beginsWith /" "phase:1,nolog,pass,t:none,id:10000,ctl:ruleRemoveById=960015"
```

We define a rule that first inspects the path. With the condition for the path *"/"*, the rule will of course always apply and the condition is therefore not needed. We prefer to set it this way nevertheless, because this base pattern can easily be used for different paths. Without a condition we would formulate it as *SecAction*. In phase 1 of our rule we define that we don't want to log, but assign an ID to the beginning of our block (*10000*). Finally, we suppress rule *960015*. This is done via a control instruction (*ctl.*).

That was very important. Therefore, to summarize once again: We define a rule to suppress another rule. We use a pattern for this which lets us define a path as a condition. This enables us to disable rules for individual parts of an application. Only in those places where false alarms occur. This prevents us from disabling rules on the entire server, considering that the false alarm occurs only when processing one individual form, which is frequently the case. This would look something like this:

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" "phase:1,nolog,pass,t:none,id:10001,\
ctl:ruleRemoveById=960015"
```

We have now disabled a rule. For the entire service (*"/"*) or for a specific sub-path (*"/app/submit.do"*). Unfortunately, we have gone a bit blind with respect to these rules. We never know whether incoming requests would violate the rule. Because we aren't always familiar with the applications on our servers in detail and if we now wait a year and think about whether we still need the *ignore rule*, we won't have an answer for that. We have suppressed every message on the topic. It would be ideal if we could still observe when the rule takes effect, but without blocking the request and having the *anomaly score* remain unchanged. Increasing an *anomaly score* is done in the definition of the rule. In rule *960015* of the *core rules* this is solved as follows:

```
setvar:tx.inbound_anomaly_score=+%{tx.notice_anomaly_score}"
```

Here the value *tx.notice_anomaly_score* is added to the *inbound_anomaly_score* transaction variable. We have the option of changing the configuration of this rule without touching the rule file. We can't suppress addition, but we can neutralize it by subtraction. However, this means another rule pattern in the form of a rule configured after the *core rules* are embedded.

```
...
SecRule REQUEST_FILENAME "@beginsWith /index.html" "chain,phase:2,log,pass,t:none,id:10004,\
    msg:'Adjusting inbound anomaly score for rule 960015'"
   SecRule "&TX:960015-OWASP_CRS/PROTOCOL_VIOLATION/MISSING_HEADER-REQUEST_HEADERS" "@ge 1" \
    "setvar:tx.inbound_anomaly_score=-%{tx.notice_anomaly_score}"
...
```

Here we have two rules combined via the *chain* command. This means that the first rule formulates a condition and the second rule is only executed if the first condition is true. Another somewhat cryptic condition is formulated in the second rule. Specifically, we are looking if a specific variable is set, in particular *TX:960015-OWASP_CRS/PROTOCOL_VIOLATION/MISSING_HEADER-REQUEST_HEADERS*. This transaction variable was set by

rule *960015* and indicates a match for rule 960015. So, if we find this variable this means that rule 960015 was applied. In this case we again reduce the *inbound anomaly score* by the same value the rule increased it. We thus neutralize the effect of the rule without suppressing the message itself.

Afterwards, for the *curl* request presented above this results in two entries in the *error log*:

```
[2015-11-08 08:16:19.215089] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Operator EQ matched 0 a
[2015-11-08 08:16:19.220263] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Operator GE matched 1 a
```

The entry about the final *anomaly score* does not appear, because this value was reset to 0. All this means that we will now continue to be informed if a rule violation occurs, but the request will no longer be assigned an *anomaly score* based on this rule. In this sense, we have accepted the alarm.

These types of rules are demanding: While the first condition follows a familiar pattern, the second rule which includes the transaction variable involves a lot of writing: How do we get to the variable names and from where exactly do we know the score?

We can either get the variable names from the rule definitions in the *core rules* or stick with the *debug log* that we define at the highest level (*SecDebugLogLevel 9*):

```
$> sudo egrep "Set variable.*960015" logs/modsec_debug.log
[16/Dec/2013:10:16:11 +0100] [localhost/sid#1470170][rid#7fbc5c018e40][/app/submit.do][9] Set variable "tx.
```

The *tx.960015-OWASP_CRS/PROTOCOL_VIOLATION/MISSING_HEADER-REQUEST_HEADERS* written here must be written in a rule as *TX:960015-OWASP_CRS/PROTOCOL_VIOLATION/MISSING_HEADER-REQUEST_HEADERS*, written exactly as we just did above. The *&* in front means that it's not the variable itself being inspected, but the number of variables with this name: *1*. The exact value by which we have to subtract again in the *inbound anomaly score* is again found in the *debug log*:

```
$> sudo egrep -B9 "Set variable.*960015" logs/modsec_debug.log
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Setting variable:
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Recorded original
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Resolved macro %{t
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Relative change: a
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Set variable "tx.a
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Setting variable:
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Resolved macro %{r
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Resolved macro %{m
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Resolved macro %{m
[08/Nov/2015:08:16:19 +0100] [localhost/sid#758700][rid#7fee30002970][/app/submit.do][9] Set variable "tx.9
```

Here we see in detail how *ModSecurity* performed its arithmetic functions. What's interesting is the first line showing how the anomaly score is increased. It is increased by *tx.notice_anomaly_score*. We would also find this value in the definition of the *core rules*, but it's easier to read here.

It can of course happen that a rule is violated multiple times. This means that multiple parameters are violating the same rule. We can also cover this case, at yet another level of complexity. In this case, *ModSecurity* writes a collection variable that includes the variable names: rule number and name with the suffix *-ARGS:* added to it. For a *file injection rule* such as *950005* this results, for example, in *TX:950005-OWASP_CRS/WEB_ATTACK/FILE_INJECTION-ARGS:contact_form_name*. If we are dealing with two parameters that are violating this rule (say, for instance *contact_form_name* and *contact_form_address*), we use the following construct:

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" "chain,phase:2,log,t:none,pass,id:10003, \
    msg:'Adjusting inbound anomaly score for rule 950005'"
   SecRule "&TX:950005-OWASP_CRS/WEB_ATTACK/FILE_INJECTION-ARGS:contact_form_name" "@ge 1" \
    "setvar:tx.inbound_anomaly_score=-%{tx.critical_anomaly_score}"

SecRule REQUEST_FILENAME "@beginsWith /submit.do" "chain,phase:2,log,pass,t:none,id:10004, \
    msg:'Adjusting inbound anomaly score for rule 950005'"
   SecRule "&TX:950005-OWASP_CRS/WEB_ATTACK/FILE_INJECTION-ARGS:contact_form_address" "@ge 1" \
    "setvar:tx.inbound_anomaly_score=-%{tx.critical_anomaly_score}"
```

We can trigger the rule violation and change of anomaly value just configured with the following request:

```
$> curl -d "contact_form_name=/etc/passwd" -d "contact_form_address=/etc/passwd" \
    http://localhost/app/submit.do
...
$> tail -1 logs/access.log
127.0.0.1 - - [2015-11-08 08:25:46.950543] "POST /app/submit.do HTTP/1.1" 404 45 "-" \
"curl/7.46.0-DEV" localhost 127.0.0.1 80 - - "-" Vj74@n8AAQEAADydhKkAAAAE - - 219 231 \
-% 21734 19189 71 588 0 0
$> grep Vj74@n8AAQEAADydhKkAAAAE logs/error.log
...
[2015-11-08 08:25:46.961718] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Pattern match "(?:\\\\b
[2015-11-08 08:25:46.961953] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Pattern match "(?:\\\\b
[2015-11-08 08:25:46.970259] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Operator GE matched 1 a
[2015-11-08 08:25:46.970320] [-:error] - - [client 127.0.0.1] ModSecurity: Warning. Operator GE matched 1 a
```

As you see, we can instruct *ModSecurity* to apply the core rules without having to increase the score. But to be honest, you have to admit that the design of such a construct is extremely complex and error-prone. This is why I don't use this technique in practice, but only suppress *false positives* by selectively disabling rules and for this reason am struggling with the blindness described at the beginning.

There's a very simple method for disabling the rules that we are not familiar with yet:

Step 7: Suppressing false alarms: Disabling individual rules for specific parameters

Till now we have suppressed individual rules for specific paths. In practice there is a second case that at least quantitatively is very widespread: An individual parameter, typically a cookie, triggers rule violation regardless of path. Each individual request results in a rule violation. An initial look at the statistics can come as quite a shock. But only when you see how easy this is to handle does the situation settle down a bit. You'd have to disable the rule for the base path `/` or manage to generally disable the rules for the affected parameter. This is done as follows:

```
SecRuleUpdateTargetById 950005 "!REQUEST_COOKIES:basket_id"
```

This directive, that has to be configured after loading the *core rules*, matches the *target list* in rule 950005. This means that the cookie *basket_id* should no longer be inspected by rule 950005. This again results in blindness, but a cookie can be very easily checked at a later point in time as to whether the rules related to it are still relevant.

For form parameters we shouldn't proceed so generally that we disable it for the entire service. There is however another rule pattern closely based on this example, but which is only effective on a single path for an individual parameter:

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" \
    "phase:2,nolog,pass,t:none,id:10002,ctl:ruleRemoveTargetById=950005;ARGS:contact_form_name"
```

We disable the handling of the *contact_form_name* parameter via rule *950005* for the path */app/submit.do*.. This does the job right and in my experience is the preferred way to suppress an individual false positive for a parameter.

Using the different methods to design *ignore rules* gives us the tools we need to work through the *false positives* one by one. Being able to work quickly requires experience and helpful tools which we will becoming familiar with in the next tutorial.

Step 8: Readjusting the anomaly limit

You now use the patterns for *ignore rules* described above to work through the various *false positives*. This work is very complex. However, with the goal of protecting the application in detail, it is most certainly worthwhile. What should however be considered is the degree to which the *false positives* are to be eliminated. A typical target value is something such as blocking a request that violated a rule on a level lower than *critical*. This means setting the *anomaly limit* to 5. Now, every request that violates a critical rule is assigned a score of a least 5 and is blocked in the end. A service tuned this way could receive an *access log* analysis using the following pattern:

```
$> egrep  -o "[0-9]+ [0-9]+$" logs/access.log   | ./modsec-positive-stats.rb
INCOMING                    Num of req. | % of req. |  Sum of % | Missing %
```

```
Number of incoming req. (total) |  10000 | 100.0000% | 100.0000% |   0.0000%

Empty or miss. incoming score   |      0 |   0.0000% |   0.0000% | 100.0000%
Reqs with incoming score of   0 |   9970 |  99.7000% |  99.7000% |   0.3000%
Reqs with incoming score of   1 |      4 |   0.0400% |  99.7400% |   0.2600%
Reqs with incoming score of   2 |     21 |   0.2100% |  99.9500% |   0.0500%
Reqs with incoming score of   3 |      0 |   0.0000% |  99.9500% |   0.0500%
Reqs with incoming score of   4 |      4 |   0.0400% |  99.9900% |   0.0100%
Reqs with incoming score of   5 |      1 |   0.0100% | 100.0000% |   0.0000%

Average:   0.0067        Median   0.0000         Standard deviation   0.1329


OUTGOING                     Num of req. | % of req. |  Sum of % | Missing %
Number of outgoing req. (total) |  10000 | 100.0000% | 100.0000% |   0.0000%

Empty or miss. outgoing score   |      0 |   0.0000% |   0.0000% | 100.0000%
Reqs with outgoing score of   0 |  10000 | 100.0000% | 100.0000% |   0.0000%

Average:   0.0000        Median   0.0000         Standard deviation   0.0000
```

Of 10,000 requests, 9,999 have an *anomaly score* of 4 or lower. This is a fine tuned service. We can thus reduce the *inbound anomaly score limit* increased at the beginning. If we want to be notified of critical rule violations, we set the limits as follows:

```
...
SecAction "id:'900002',phase:1,t:none,setvar:tx.inbound_anomaly_score_level=5,nolog,pass"
SecAction "id:'900003',phase:1,t:none,setvar:tx.outbound_anomaly_score_level=5,nolog,pass"
...
```

We do the same with the *outbound anomaly score*. In practice, you'll have to be a bit more nuanced.

Step 9: Summary of the ways of combating false positives

We have become familiar with four different ways of suppressing a false alarm. I'm presenting them here again along with specific examples:

Case 1 : Disabling the rule for a specific path

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" \
    "phase:1,nolog,pass,t:none,id:10001,ctl:ruleRemoveById=950005"
```

Location in the configuration: Preferably in front of the *Core Rule Inclusion*, but for *phase:1* can also be placed after *include*.
Phase: Best in phase 1, because the path is known at this moment.

Case 2 : Disabling rules for specific parameters

```
SecRuleUpdateTargetById 950005 "!ARGS:contact_form_name"
```

Location in the configuration: Generally placed after the *Core Rule Inclusion*.

Case 3 : Disabling rules for specific parameters on a specific path

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" \
    "phase:2,nolog,pass,t:none,id:10002,ctl:ruleRemoveTargetById=950005;ARGS:contact_form_name"
```

Location in the configuration: Preferably placed after the *Core Rule Inclusion*.
Phase: Required for *post parameter* in phase 2, otherwise conceivable in phase 1.

Case 4 : Keep the rule enabled, but disable scoring for specific parameters on specific paths

```
SecRule REQUEST_FILENAME "@beginsWith /app/submit.do" \
    "chain,phase:2,log,pass,t:none,id:10003,msg:'Adjusting inbound anomaly score for rule 950005'"
SecRule "&TX:950005-OWASP_CRS/WEB_ATTACK/FILE_INJECTION-ARGS:contact_form_name" "@ge 1" \
    "setvar:tx.inbound_anomaly_score=-%{tx.critical_anomaly_score}"
```

Location in the configuration: Preferably placed after the *Core Rule Inclusion*.
Phase: Required for *post parameter* in phase 2, otherwise conceivable in phase 1.

In practice, it's important to proceed systematically. *ModSecurity* and the *core rules* are already complex enough. If we don't watch out, all of our fine tuning efforts will only result in chaos at the end. It's better to think about which rule fine tuning approaches you want to work with. In technical terms, the one manipulating the *anomaly scores* is the preferred approach. It is however very hard to read and the work required for writing and testing outweighs the simpler ones, although this in turn is accompanied by the disadvantage described above that all messages for the rule are suppressed.

In the next tutorial we will be turning to practice and deriving fine tuning rules for an untuned application from the log files.

Step 10 (Goodie): A beer

This tutorial was a hard bit of work. We'll be taking a break now and treating ourselves to a beer. The next tutorial turns to practice. We have become familiar with the basic techniques, but how exactly do you apply these techniques when you are sinking under a huge number of false positives?

References

- Spider Labs Blog Post: Exception Handling
- ModSecurity Reference Manual

License / Copying / Further use