ModSecurity 2 Data Formats

Version 2.5.10-dev1 (March 24, 2009)

Table of Contents

The purpose of this document is to describe the formats of the ModSecurity alert messages, transaction logs and communication protocols, which would not only allow for a better understanding what ModSecurity does but also for an easy integration with third-party tools and products.

Alerts

As part of its operations ModSecurity will emit alerts, which are either *warnings* (non-fatal) or *errors* (fatal, usually leading to the interception of the transaction in question). Below is an example of a ModSecurity alert entry:

```
Access denied with code 505 (phase 1). Match of "rx
  ^HTTP/(0\\\\.9|1\\\\.[01])$" against "REQUEST_PROTOCOL" required.
  [id "960034"] [msg "HTTP protocol version is not allowed by policy"]
  [severity "CRITICAL"] [uri "/"] [unique_id "PQaTTVBEUOkAAFwKXrYAAAAM"]
```

Note

Alerts will only ever contain one line of text but we've broken the above example into multiple lines to make it fit into the page.

Each alert entry begins with the engine message, which describes what ModSecurity did and why. For example:

```
Access denied with code 505 (phase 1). Match of "rx
  ^HTTP/(0\\\\.9|1\\\\.[01])$" against "REQUEST_PROTOCOL" required.
```

Alert Action Description

The first part of the engine message tells you whether ModSecurity acted to interrupt transaction or rule processing:

1. If the alert is only a warning, the first sentence will simply say *Warning*.
2. If the transaction was intercepted, the first sentence will begin with *Access denied*. What follows is the list of possible messages related to transaction interception:
   - *Access denied with code %0* - a response with status code `%0` was sent.
   - *Access denied with connection close* - connection was abruptly closed.
   - *Access denied with redirection to %0 using status %1* - a redirection to URI `%0` was issued using status `%1`.
3. There is also a special message that ModSecurity emits where an `allow` action is executed. There are three variations of this type of message:
   - *Access allowed* - rule engine stopped processing rules (transaction was unaffected).
   - *Access to phase allowed* - rule engine stopped processing rules in the current phase only. Subsequent phases will be processed normally. Transaction was not affected by this rule but it may be affected by any of the rules in the subsequent phase.
   - *Access to request allowed* - rule engine stopped processing rules in the current phase. Phases prior to request execution in the backend (currently phases 1 and 2) will not

be processed. The response phases (currently phases 3 and 4) and others (currently phase 5) will be processed as normal. Transaction was not affected by this rule but it may be affected by any of the rules in the subsequent phase.

Alert Justification Description

The second part of the engine message explains *why* the alert was generated. Since it is automatically generated from the rules it will be very technical in nature, talking about operators and their parameters and give you insight into what the rule looked like. But this message cannot give you insight into the reasoning behind the rule. A well-written rule will always specify a human-readable message (using the `msg` action) to provide further information.

The format of the second part of the engine message depends on whether it was generated by the operator (which happens on a match) or by the rule processor (which happens where there is not a match, but the negation was used):

- `@beginsWith` - *String match %0 at %1.*
- `@contains` - *String match %0 at %1.*
- `@containsWord` - *String match %0 at %1.*
- `@endsWith` - *String match %0 at %1.*
- `@eq` - *Operator EQ matched %0 at %1.*
- `@ge` - *Operator GE matched %0 at %1.*
- `@geoLookup` - *Geo lookup for %0 succeeded at %1.*
- `@inspectFile` - *File %0 rejected by the approver script %1: %2*
- `@le` - *Operator LE matched %0 at %1.*
- `@lt` - *Operator LT matched %0 at %1.*
- `@rbl` - *RBL lookup of %0 succeeded at %1.*
- `@rx` - *Pattern match %0 at %1.*
- `@streq` - *String match %0 at %1.*
- `@validateByteRange` - *Found %0 byte(s) in %1 outside range: %2.*
- `@validateDTD` - *XML: DTD validation failed.*
- `@validateSchema` - *XML: Schema validation failed.*
- `@validateUrlEncoding`
  - *Invalid URL Encoding: Non-hexadecimal digits used at %0.*
  - *Invalid URL Encoding: Not enough characters at the end of input at %0.*
- `@validateUtf8Encoding`
  - *Invalid UTF-8 encoding: not enough bytes in character at %0.*
  - *Invalid UTF-8 encoding: invalid byte value in character at %0.*
  - *Invalid UTF-8 encoding: overlong character detected at %0.*

    - *Invalid UTF-8 encoding: use of restricted character at %0.*
    - *Invalid UTF-8 encoding: decoding error at %0.*
  - `@verifyCC` - *CC# match %0 at %1.*

Messages not related to operators:

- When `SecAction` directive is processed - *Unconditional match in SecAction.*
- When `SecRule` does not match but negation is used - *Match of %0 against %1 required.*

Note

The parameters to the operators `@rx` and `@pm` (regular expression and text pattern, respectively) will be truncated to 252 bytes if they are longer than this limit. In this case the parameter in the alert message will be terminated with three dots.

Meta-data

The metadata fields are always placed at the end of the alert entry. Each metadata field is a text fragment that consists of an open bracket followed by the metadata field name, followed by the value and the closing bracket. What follows is the text fragment that makes up the `id` metadata field.

```
[id "960034"]
```

The following metadata fields are currently used:

1. `offset` - The byte offset where a match occured within the target data. This is not always available.
2. `id` - Unique rule ID, as specified by the `id` action.
3. `rev` - Rule revision, as specified by the `rev` action.
4. `msg` - Human-readable message, as specified by the `msg` action.
5. `severity` - Event severity as text, as specified by the `severity` action. The possible values (with their corresponding numberical values in brackets) are `EMERGENCY` (0), `ALERT` (1), `CRITICAL` (2), `ERROR` (3), `WARNING` (4), `NOTICE` (5), `INFO` (6) and `DEBUG` (7).
6. `unique_id` - Unique event ID, generated automatically.
7. `uri` - Request URI.
8. `logdata` - contains transaction data fragment, as specified by the `logdata` action.

Escaping

ModSecurity alerts will always contain text fragments that were taken from configuration or the transaction. Such text fragments escaped before they are user in messages, in order to sanitise

the potentially dangerous characters. They are also sometimes surrounded using double quotes. The escaping algorithm is as follows:

1. Characters `0x08` (`BACKSPACE`), `0x0a` (`NEWLINE`), `0x10` (`CARRIAGE RETURN`), `0x09` (`HORIZONTAL TAB`) and `0x0b` (`VERTICAL TAB`) will be represented as `\b`, `\n`, `\r`, `\t` and `\v`, respectively.
2. Bytes from the ranges `0-0x1f` and `0x7f-0xff` (inclusive) will be represented as `\xHH`, where `HH` is the hexadecimal value of the byte.
3. Backslash characters (`\`) will be represented as `\\`.
4. Each double quote character will be represented as `\"`, but only if the entire fragment is surrounded with double quotes.

Alerts in the Apache Error Log

Every ModSecurity alert conforms to the following format when it appears in the Apache error log:

```
[Sun Jun 24 10:19:58 2007] [error] [client 192.168.0.1]
            ModSecurity: ALERT_MESSAGE
```

The above is a standard Apache error log format. The `ModSecurity:` prefix is specific to ModSecurity. It is used to allow quick identification of ModSecurity alert messages when they appear in the same file next to other Apache messages.

The actual message (`ALERT_MESSAGE` in the example above) is in the same format as described in the *Alerts* section.

Note

Apache further escapes ModSecurity alert messages before writing them to the error log. This means that all backslash characters will be doubled in the error log. In practice, since ModSecurity will already represent a single backslash within an untrusted text fragment as two backslashes, the end result in the Apache error log will be *four* backslashes. Thus, if you need to interpret a ModSecurity message from the error log, you should decode the message part after the `ModSecurity:` prefix first. This step will peel the first encoding layer.

Alerts in Audit Logs

Alerts are transported in the `H` section of the ModSecurity Audit Log. Alerts will appear each on a separate line and in the order they were generated by ModSecurity. Each line will be in the following format:

```
Message: ALERT_MESSAGE
```

Below is an example of an H section that contains two alert messages:

```
--c7036611-H--
Message: Warning. Match of "rx ^apache.*perl" against
  "REQUEST_HEADERS:User-Agent" required. [id "990011"] [msg "Request
  Indicates an automated program explored the site"] [severity "NOTICE"]
Message: Warning. Pattern match "(?:\\b(?:(?:s(?:elect\\b(?:.{1,100}?\\b
  (?:(?:length|count|top)\\b.{1,100}?\\bfrom|from\\b.{1,100}?\\bwhere)
  |.*?\\b(?:d(?:ump\\b.*\\bfrom|ata_type)|(?:to_(?:numbe|cha)|inst)r))|p_
  (?:(?:addextendedpro|sqlexe)c|(?:oacreat|prepar)e|execute(?:sql)?|
  makewebt ..." at ARGS:c. [id "950001"] [msg "SQL Injection Attack.
  Matched signature: union select"] [severity "CRITICAL"]
Stopwatch: 1199881676978327 2514 (396 2224 -)
Producer: ModSecurity v2.x.x (Apache 2.x)
Server: Apache/2.x.x

--c7036611-Z--
```

Audit Log

ModSecurity records one transaction in a single audit log file. Below is an example:

```
--c7036611-A--
[09/Jan/2008:12:27:56 +0000] OSD4l1BEUOkAAHZ8Y3QAAAAH 209.90.77.54 64995
  80.68.80.233 80
--c7036611-B--
GET //EvilBoard_0.1a/index.php?c='/**/union/**/select/**/1,concat(username,
  char(77),password,char(77),email_address,char(77),info,char(77),user_level,
  char(77))/**/from/**/eb_members/**/where/**/userid=1/*http://kamloopstutor.
  com/images/banners/on.txt? HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, cslose
Host: www.example.com
User-Agent: libwww-perl/5.808

--c7036611-F--
HTTP/1.1 404 Not Found
Content-Length: 223
Connection: close
Content-Type: text/html; charset=iso-8859-1

--c7036611-H--
Message: Warning. Match of "rx ^apache.*perl" against
  "REQUEST_HEADERS:User-Agent" required. [id "990011"] [msg "Request
  Indicates an automated program explored the site"] [severity "NOTICE"]
Message: Warning. Pattern match "(?:\\b(?:(?:s(?:elect\\b(?:.{1,100}?\\b
  (?:(?:length|count|top)\\b.{1,100}?\\bfrom|from\\b.{1,100}?\\bwhere)
  |.*?\\b(?:d(?:ump\\b.*\\bfrom|ata_type)|(?:to_(?:numbe|cha)|inst)r))|p_
  (?:(?:addextendedpro|sqlexe)c|(?:oacreat|prepar)e|execute(?:sql)?|
  makewebt ..." at ARGS:c. [id "950001"] [msg "SQL Injection Attack.
  Matched signature: union select"] [severity "CRITICAL"]
Stopwatch: 1199881676978327 2514 (396 2224 -)
Producer: ModSecurity v2.x.x (Apache 2.x)
Server: Apache/2.x.x

--c7036611-Z--
```

The file consist of multiple sections, each in different format. Separators are used to define sections:

```
--c7036611-A--
```

A separator always begins on a new line and conforms to the following format:

1. Two dashes
2. Unique boundary, which consists from several hexadecimal characters.
3. One dash character.
4. Section identifier, currently a single uppercase letter.
5. Two trailing dashes.

Refer to the documentation for `SecAuditLogParts` for the explanation of each part.

Parts

This section documents the audit log parts available in ModSecurity 2.x. They are:

- `A` - audit log header
- `B` - request headers
- `C` - request body
- `D` - intended response headers (NOT IMPLEMENTED)
- `E` - intended response body
- `F` - response headers
- `G` - response body (NOT IMPLEMENTED)
- `H` - audit log trailer
- `I` - reduced multipart request body
- `J` - multipart files information (NOT IMPLEMENTED)
- `K` - matched rules information
- `Z` - audit log footer

Audit Log Header (A)

ModSecurity 2.x audit log entries always begin with the header part. For example:

```
--c7036611-A--
[09/Jan/2008:12:27:56 +0000] OSD4l1BEUOkAAHZ8Y3QAAAAH 209.90.77.54 64995
  80.68.80.233 80
```

The header contains only one line, with the following information on it:

1. Timestamp
2. Unique transaction ID
3. Source IP address (IPv4 or IPv6)
4. Source port
5. Destination IP address (IPv4 or IPv6)

6. Destination port

Request Headers (B)

The request headers part contains the request line and the request headers. The information present in this part will not be identical to that sent by the client responsible for the transaction. ModSecurity 2.x for Apache does not have access to the raw data; it sees what Apache itself sees. While the end result may be identical to the raw request, differences are possible in some areas:

1. If any of the fields are `NUL`-terminated, Apache will only see the content prior to the NUL.
2. Headers that span multiple lines (feature known as header folding) will be collapsed into a single line.
3. Multiple headers with the same name will be combined into a single header (as allowed by the HTTP RFC).

Request Body (C)

This part contains the request body of the transaction, after dechunking and decompression (if applicable).

Intended Response Headers (D)

This part contains the status line and the request headers that would have been delivered to the client had ModSecurity not intervened. Thus this part makes sense only for transactions where ModSecurity altered the data flow. By differentiating before the intended and the final response headers, we are able to record what was internally ready for sending, but also what was actually sent.

Note

This part is reserved for future use. It is not implemented in ModSecurity 2.x.

Intended Response Body (E)

This part contains the transaction response body (before compression and chunking, where used) that was either sent or would have been sent had ModSecurity not intervened. You can find whether interception took place by looking at the `Action` header of the part `H`. If that header is present, and the interception took place in phase 3 or 4 then the `E` part contains the intended response body. Otherwise, it contains the actual response body.

Note

Once the `G` (actual response body) part is implemented, part `E` will be present only in audit logs that contain a transaction that was intercepted, and there will be no need for further analsys.

Response Headers (F)

This part contains the actual response headers sent to the client. Since ModSecurity 2.x for Apache does not access the raw connection data, it constructs part `F` out of the internal Apache data structures that hold the response headers.

Some headers (the `Date` and `Server` response headers) are generated just before they are sent and ModSecurity is not able to record those. You should note than ModSecurity is working as part of a reverse proxy, the backend web server will have generated these two servers, and in that case they will be recorded.

Response Body (G)

When implemented, this part will contain the actual response body before compression and chunking.

Note

This part is reserved for future use. It is not implemented in ModSecurity 2.x.

Audit Log Trailer (H)

Part `H` contains additional transaction meta-data that was obtained from the web server or from ModSecurity itself. The part contains a number of trailer headers, which are similar to HTTP headers (without support for header folding):

1. Action
2. Apache-Error
3. Message
4. Producer
5. Response-Body-Transformed
6. Sanitised-Args
7. Sanitised-Request-Headers
8. Sanitised-Response-Headers
9. Server
10. Stopwatch
11. WebApp-Info

Action

The `Action` header is present only for the transactions that were intercepted:

```
Action: Intercepted (phase 2)
```

The phase information documents the phase in which the decision to intercept took place.

Apache-Error

The Apache-Error header contains Apache error log messages observed by ModSecurity, excluding those sent by ModSecurity itself. For example:

```
Apache-Error: [file "/tmp/buildd/apache2-2.0.54/build-tree/apache2/server/
  core.c"] [line 3505] [level 3] File does not exist: /var/www/www.
  modsecurity.org/fst/documentation/modsecurity-apache/2.5.0-dev2
```

Message

Zero or more `Message` headers can be present in any trailer, and each such header will represent a single ModSecurity warning or error, displayed in the order they were raised.

The example below was broken into multiple lines to make it fit this page:

```
Message: Access denied with code 400 (phase 2). Pattern match "^\w+:/" at
  REQUEST_URI_RAW. [file "/etc/apache2/rules-1.6.1/modsecurity_crs_20_
  protocol_violations.conf"] [line "74"] [id "960014"] [msg "Proxy access
  attempt"] [severity "CRITICAL"] [tag "PROTOCOL_VIOLATION/PROXY_ACCESS"]
```

Producer

The `Producer` header identifies the product that generated the audit log. For example:

```
Producer: ModSecurity for Apache/2.5.5 (http://www.modsecurity.org/).
```

ModSecurity allows rule sets to add their own signatures to the `Producer` information (this is done using the `SecComponentSignature` directive). Below is an example of the `Producer` header with the signature of one component (all one line):

```
Producer: ModSecurity for Apache/2.5.5 (http://www.modsecurity.org/);
    MyComponent/1.0.0 (Beta).
```

Response-Body-Transformed

This header will appear in every audit log that contains a response body:

```
Response-Body-Transformed: Dechunked
```

The contents of the header is constant at present, so the header is only useful as a reminder that the recorded response body is not identical to the one sent to the client. The actual content is the same, except that Apache may further compress the body and deliver it in chunks.

Sanitised-Args

The `Sanitised-Args` header contains a list of arguments that were sanitised (each byte of their content replaced with an asterisk) before logging. For example:

```
Sanitised-Args: "old_password", "new_password", "new_password_repeat".
```

Sanitised-Request-Headers

The `Sanitised-Request-Headers` header contains a list of request headers that were sanitised before logging. For example:

```
Sanitised-Request-Headers: "Authentication".
```

Sanitised-Response-Headers

The `Sanitised-Response-Headers` header contains a list of response headers that were sanitised before logging. For example:

```
Sanitised-Response-Headers: "My-Custom-Header".
```

Server

The `Server` header identifies the web server. For example:

```
Server: Apache/2.0.54 (Debian GNU/Linux) mod_ssl/2.0.54 OpenSSL/0.9.7e
```

This information may sometimes be present in any of the parts that contain response headers, but there are a few cases when it isn't:

1. None of the response headers were recoreded.
2. The information in the response headers is not accurate because server signature masking was used.

Stopwatch

The `Stopwatch` header provides certain diagnostic information that allows you to determine the performance of the web server and of ModSecurity itself. It will typically look like this:

```
Stopwatch: 1222945098201902 2118976 (770* 4400 -)
```

Each line can contain up to 5 different values. Some values can be absent; each absent value will be replaced with a dash.

The meanings of the values are as follows (all values are in microseconds):

1. Transaction timestamp in microseconds since January 1st, 1970.
2. Transaction duration.
3. The time between the moment Apache started processing the request and until phase 2 of ModSecurity began. If an asterisk is present that means the time includes the time it took ModSecurity to read the request body from the client (typically slow). This value can be used to provide a rough estimate of the client speed, but only with larger request bodies (the smaller request bodies may arrive in a single TCP/IP packet).

4. The time between the start of processing and until phase 2 was completed. If you substract the previous value from this value you will get the exact duration of phase 2 (which is the main rule processing phase).
5. The time between the start of request processing and util we began sending a fully-buffered response body to the client. If you substract this value from the total transaction duration and divide with the response body size you may get a rough estimate of the client speed, but only for larger response bodies.

WebApp-Info

The `WebApp-Info` header contains information on the application to which the recorded transaction belongs. This information will appear only if it is known, which will happen if `SecWebAppId` was set, or `setsid` or `setuid` executed in the transaction.

The header uses the following format:

```
WebApp-Info: "WEBAPPID" "SESSIONID" "USERID"
```

Each unknown value is replaced with a dash.

Reduced Multipart Request Body (`I`)

Transactions that deal with file uploads tend to be large, yet the file contents is not always relevant from the security point of view. The `I` part was designed to avoid recording raw `multipart/form-data` request bodies, replacing them with a simulated `application/x-www-form-urlencoded` body that contains the same key-value parameters.

The reduced multipart request body will not contain any file information. The `J` part (currently not implemented) is intended to carry the file metadata.

Multipart Files Information (`J`)

The purpose of part `J` is to record the information on the files contained in a `multipart/form-data` request body. This is handy in the cases when the original request body was not recorded, or when only a reduced version was recorded (e.g. when part `I` was used instead of part `C`).

Note

This part is reserved for future use. It is not implemented in ModSecurity 2.x.

Matched Rules (`K`)

The matched rules part contains a record of all ModSecurity rules that matched during transaction processing. You should note that if a rule that belongs to a chain matches then the entire chain

will be recorded. This is because, even though the disruptive action may not have executed, other per-rule actions have, and you will need to see the entire chain in order to understand the rules.

This part is available starting with ModSecurity 2.5.x.

Audit Log Footer (Z)

Part Z is a special part that only has a boundary but no content. Its only purpose is to signal the end of an audit log.

Storage Formats

ModSecurity supports two audit log storage formats:

1. *Serial* audit log format - multiple audit log files stored in the same file.
2. *Concurrent* audit log format - one file is used for every audit log.

Serial Audit Log Format

The serial audit log format stores multiple audit log entries within the same file (one after another). This is often very convinent (audit log entries are easy to find) but this format is only suitable for light logging in the current ModSecurity implementation because writing to the file is serialised: only one audit log entry can be written at any one time.

Concurrent Audit Log Format

The concurrent audit log format uses one file per audit log entry, and allows many transactions to be recorded at once. A hierarchical directory structure is used to ensure that the number of files created in any one directory remains relatively small. For example:

```
$LOGGING-HOME/20081128/20081128-1414/20081128-141417-
  egDKy38AAAEAAAyMHXsAAAAA
```

The current time is used to work out the directory structure. The file name is constructed using the current time and the transaction ID.

The creation of every audit log in concurrent format is recorded with an entry in the concurrent audit log *index file*. The format of each line resembles the common web server access log format. For example:

```
192.168.0.111 192.168.0.1 - - [28/Nov/2008:15:06:32 +0000]
  "GET /?p=\\ HTTP/1.1" 200 69 "-" "-" NOfRx38AAAEAAAzcCU4AAAAA
  "-" /20081128/20081128-1506/20081128-150632-NOfRx38AAAEAAAzcCU4AAAAA
  0 1183 md5:ffee2d414cd43c2f8ae151652910ed96
```

The tokens on the line are as follows:

1. Hostname (or IP address, if the hostname is not known)

2. Source IP address
3. Remote user (from HTTP Authentication)
4. Local user (from identd)
5. Timestamp
6. Request line
7. Response status
8. Bytes sent (in the response body)
9. Referrer information
10. User-Agent information
11. Transaction ID
12. Session ID
13. Audit log file name (relative to the audit logging home, as configured using the `SecAuditLogStorageDir` directive)
14. Audit log offset
15. Audit log size
16. Audit log hash (the has begins with the name of the algorithm used, followed by a colon, followed by the hexadecimal representation of the hash itself); this hash can be used to verify that the transaction was correctly recorded and that it hasn't been modified since.

Note

Lines in the index file will be up to 3980 bytes long, and the information logged will be reduced to fit where necessary. Reduction will occur within the individual fields, but the overall format will remain the same. The character `L` will appear as the last character on a reduced line. A space will be the last character on a line that was not reduced to stay within the limit.

Transport Protocol

Audit logs generated in multi-sensor deployments are of little use if left on the sensors. More commonly, they will be transported to a central logging server using the transport protocol described in this section:

1. The transport protocol is based on the HTTP protocol.
2. The server end is an SSL-enabled web server with HTTP Basic Authentication configured.
3. Clients will open a connection to the centralisation web server and authenticate (given the end-point URI, the username and the password).

4. Clients will submit every audit log in a single `PUT` transaction, placing the file in the body of the request and additional information in the request headers (see below for details).
5. Server will process each submission and respond with an appropriate status code:
   a. 200 (OK) - the submission was processed; the client can delete the corresponding audit log entry if it so desires. The same audit log entry must not be submitted again.
   b. 409 (Conflict) - if the submission is in invalid format and cannot be processed. The client should attempt to fix the problem with the submission and attempt delivery again at a later time. This error is generally going to occur due to a programming error in the protocol implementation, and not because of the content of the audit log entry that is being transported.
   c. 500 (Internal Server Error) - if the server was unable to correctly process the submission, due to its own fault. The client should re-attempt delivery at a later time. A client that starts receiving 500 reponses to all its submission should suspend its operations for a period of time before continuing.

Note

Server implementations are advised to accept all submissions that correctly implement the protocol. Clients are unlikely to be able to overcome problems within audit log entries, so such problems are best resolved on the server side.

Note

When en error occurs, the server may place an explanation of the problem in the text part of the response line.

Request Headers Information

Each audit log entry submission must contain additional information in the request headers:

1. Header `X-Content-Hash` must contain the audit log entry hash. Clients should expect the audit log entries to be validated against the hash by the server.
2. Header `X-ForensicLog-Summary` must contain the entire concurrent format index line.
3. The `Content-Lenght` header must be present and contain the length of the audit log entry.