IronBee Reference Manual

Brian Rectanus
Ivan Ristic

IronBee Reference Manual

by Brian Rectanus and Ivan Ristic

Table of Contents

List of Examples

Preface

IronBee is a universal web application security framework. Think of IronBee as a framework for developing a system for securing web applications - a framework for building a web application firewall (WAF) if you will.

IronBee was conceived and implemented by both the original author (Ivan Ristić) and maintainer (Brian Rectanus) of the popular open source WAF, ModSecurity (www.modsecurity.org [https://www.modsecurity.org/]), however, ModSecurity is now run by the TrustWave SpiderLabs team and in no way associated with IronBee. Because of the authors' previous work on ModSecurity, there are some similarities in IronBee concepts, however IronBee was built with an entirely different set of goals. More on the similarities and difference between ModSecurity and IronBee can be found in Appendix C.

IronBee started in July 2010 through the generous support of Qualys, Inc (www.qualys.com [https://www.qualys.com/]). The idea was to build a highly portable web application security framework for the community with a license that was compatible with business. IronBee was not intended to be a complete WAF, but rather, be a framework for building a WAF. But what does this mean? Essentially IronBee gives you some basic WAF functionality, but leaves most of this open to be extended or replaced by something more tailored to your environment.

As a framework, IronBee's goals are to:

- Be minimalistic in what is contained in the core engine.

  The IronBee core engine does essentially nothing but expose an API and an event subsystem. There is no rule language, there is no inspection and there is only minimal processing of the HTTP transaction cycle. Everything else is left to modules. Load only the modules that provide functionality that is required.

- Provide an simple API for data acquisition to allow embedding IronBee in existing and future products.

  Data acquisition is not a part of the IronBee engine. Instead, the engine exposes an API to allow data to be fed to IronBee in various ways. Data acquisition is done through server plugins, which must load the IronBee engine and pass it data. This leaves IronBee open to being embedded in nearly anything that can pass HTTP data to IronBee. Currently IronBee works with the following, but many others are possible:

  - Apache Web Server 2.x
  - Apache TrafficServer 3.x
  - Nginx
  - Command Line Utility (clipp)

- Provide an API to allow extension through external modules.

  To IronBee, modules are everything. Modules utilize the core engine to implement additional functionality. Currently modules can be written in C, C++ or Lua, but extending

this to other languages is possible as well. What follow is a list of what has been implemented, but anything is possible:

- Configuration
- Parsing and exposing fields for inspection and logging
- Implementing operators, transformations and actions which other modules and rules/signatures can utilize
- Rule/signature languages
- Inspection modules
- Content transformation and modification
- Logging

- Not tie inspection to a fixed rule/signature language, but rather, allow this to be implemented by modules.

  Different types of inspection require different means. Because of this, IronBee only exposes a generic rule execution engine, but does not dictate the language in which rules are written. Currently, inspection can be performed in three different ways:

  - Write an ironbee module that hooks into various events generated by the engine. This yields full control, but requires more effort, however this effort is substantially reduced by writing Lua modules.
  - Use Lua rules. This is slightly less complex that Lua modules. Instead of using the module API, it utilizes the rule engine for execution. This allows the full power of the Lua scripting language, without the complexity of writing a full IronBee module.
  - Use the IronBee Rule Language. This is a simple language that allows you to write inspection rules (more signatures) against any data fields exposed by other modules. The current language is similar to the ModSecurity Rule Language, but has only a subset of functionality. The rule language is limited by design and allows for simple pattern matches against data fields to trigger actions.

- Not limit interaction with external systems.

  Being a framework, IronBee is meant to be embedded and interaction with other systems. Because of this, IronBee does not dictate how it is managed. There is no facility for managing configuration, rules or logs. This is left up to the administrator and/or product that ironbee is embedded in. This does not mean that IronBee does not provide some means for these things, but rather it does not dictate them. While this may seem limiting, it allows for great flexibility in how IronBee is managed. It is certainly possible to replace the file based configuration with one that interacts with a database, to manage rules from an external system and/or write logs to something other than the filesystem.

About Qualys

Qualys Inc. (NASDAQ: QLYS), is a pioneer and leading provider of cloud security and compliance solutions with over 6,000 customers in more than 100 countries, including a majority of each of the Forbes Global 100 and Fortune 100. The QualysGuard Cloud Platform and integrated suite of solutions helps organizations simplify security operations and lower the cost of compliance by delivering critical security intelligence on demand and automating the full spectrum of auditing, compliance and protection for IT systems and web applications. Founded in 1999, Qualys has established strategic partnerships with leading managed service providers and consulting organizations, including Accuvant, BT, Dell SecureWorks, Fujitsu, NTT, Symantec, Verizon, and Wipro. The company is also a founding member of the Cloud Security Alliance (CSA).

For more information, please visit www.qualys.com [https://www.qualys.com/].

Chapter 1: Introduction

IronBee is a framework and library for web application security inspection - for building a Web Application Firewall (WAF). The following components make up the IronBee framework:

- IronBee Library: The primary component containing the inspection engine.
- IronBee Modules: Extensions to the inspection engine in the form of loadable shared libraries defined by the IronBee configuration.
- Server Components: The server is the executable driving the inspection engine. This can be a standard executable, such as the command line tool, or in the form of a plugin such as the Apache trafficserver plugin and Apache httpd module.
- Configuration: There are two sets of configuration. The IronBee library must be loaded and bootstrapped via the native server configuration (for example, a commandline option, trafficserver plugin configuration or httpd module configuration). In addition, the IronBee Library must be configured via its own configuration file(s).

Software Installation

IronBee source is available from github.

https://github.com/ironbee/ironbee

The source follows standard build conventions, typically:

```
./configure --prefix=/usr/local/ironbee
make
make check
sudo make install
```

Various options are available for customization through the configure script. A full list of these options are available with:

```
./configure --help
```

See Chapter 8 for more detailed instructions on building and installing IronBee.

Configuration

First, the server component must be configured to load the IronBee library and its configuration. This is done through the server's native configuration and differs for each server. See the Chapter 2 for more specific details in configuring the server component. Once the server component is loaded, IronBee configuration consists of one or more configuration files which IronBee will read. IronBee configuration file examples are listed in the Appendix A section.

Inspection

The IronBee library is fed data by the server. This server could be a command line tool (see clipp), or through an HTTP web server or proxy. IronBee will expose data via fields. A data field specifies a name, type and value which can then be used for inspection via rules. Rules can be simple signatures against fields, a chained set of operations, stream based matchers or even lua scripts. Each rule can inspect fields and perform actions.

Data Fields

IronBee supports two primary types of data fields:

- *Scalars*, which can contain data of various types (strings, numbers or streams)
- *Collections*, which contain zero or more scalars

As fields are built from external data (parsed HTTP traffic), field names and values can contain any binary data. While the names of fields are not restricted, by convention the names of built-in fields are written using all uppercase letters.

Addressing Fields

How you use fields depends on the context. The rules, for example, typically accept one or more data fields, which means that you can give a rule a scalar field (e.g., *REQUEST_URI*) or a collection field (e.g., *ARGS*). When you give a rule a collection, it will extract all scalar fields and utilize them as if they were each specified separately.

Collections can be used as a whole, or with a selector applied. A selector is applied with the colon operator. For example, you may want to extract all `username` arguments as follows:

```
ARGS:username
```

It is important to note that selectors are filters and what is returned is a list of values which may have zero or more fields. In addition to using names as selectors, you also use regular expression patterns as selectors by bracketing a regular expression with backslashes. IronBee uses the Perl Compatible Regular Expression (PCRE) syntax. For example, the following selects all fields whose names begin with `"user"`.

```
ARGS:/^user/
```

If a field name contains unusual characters (e.g., colon, whitespace, etc.), you can quote the entire name using double quotes and then use most characters without fear of breaking anything. The first colon is used as the delimiter.

```
"ARGS:my strange:name"
```

Field Transformations

Field data often may need to be transformed or normalized before it can be used. This is accomplished via transformations. These transformations can be applied to individual fields or to all fields within a rule. For appying transformations to all fields in a rule, see the `t` rule modifier, which is essentially a shorthand notation for applying the same transformation to each individual field. For individual fields, this is done via the dot syntax with parens.

```
ARGS.lowercase()
ARGS:name.lowercase()
```

In addition to manipulating the field value, a transformation may also change the field type. For example, from a collection or string to a number.

```
# Collection count: Collection to number
ARGS.count()

# Field data length: String to number
REQUEST_HEADERS:User-Agent.length()
```

Field Expansion

In addition to using fields as input to rules, sometimes there's a need to use a field in text strings, such as adding raw values for logging via the `logdata` rule modifier. This can be achieved using field expansion. For example:

```
logdata:%{NAME}
setvar:MY_COLLECTION:%{NAME}=1
```

If the expression resolves to only one variable, the entire `%{NAME}` expression will be replaced with the field value.

> **Caution**
>
> What if the field is not a scalar? Perhaps the value could be JSON or similar format?

Rules

IronBee currently defines three types of rules. There is a basic pattern matching rule language, a more limited streaming version of the pattern matching rule language, as well as the ability to specify more complex rules which syntax is processed external to the configuration file. Currently the only external rule type is via the Lua scripting language, but more may be handled in the future. In addition to external rules, the rule processing engine and configuration syntax are decoupled, allowing modules to be developed to provide alternate custom rules which interact with the same rule execution engine.

Basic Matching Rules

Basic matching rules are configured via the `Rule` directive. These rules include a list of fields containing the data to be inspected, an operator with parameter to perform the inspection, and modifiers which specify metadata attributes as well as any actions to be taken.

```
Rule REQUEST_HEADERS ARGS @rx "Some.*Pattern" id:1 rev:1 phase:REQUEST event block:phase
```

Basic matching rules will iterate through the list of fields (and sub-fields within collections), executing the specified operator and performing any required actions. Currently, the order in which the rule executes depends on both the specified phase as well as the order in which the rule is specified in the configuration.

The phase information, assigned to the rule via the phase modifier, determines when a rule will run within transaction lifecycle. Within a phase, configuration determines how rules are ordered. When a rule is read from the configuration files, it is appended to the list of rules in the desired phase. At run-time, the engine will process all of the rules one by one until interrupted.

Stream Matching Rules

While the basic matching rules are quite flexible, they are limited to executing only once in the given phase. With this limitation, you can only inspect data that is available at the time of execution. To do this effectively, the data must be buffered so that it can all be inspected in a single pass. Streaming inspection allows you to avoid buffering potentially large amounts of data by inspecting the data in smaller chunks. With this, however, comes restrictions.

The StreamInspect directive allows inspecting a limited set of fields (currently only the raw request and response bodies as of version 0.7) in smaller chunks as the data arrives. Instead of the rule executing only a single time, it may instead execute many times - once for each chunk of data. Because of this, stream based rules do not have a phase associated with them. In addition to this difference from the basic matching rules, stream based rules cannot (currently) be transformed and allow only a limited set of operators (currently `dfa`, `pm` and `pmf` as of version 0.7).

```
StreamInspect REQUEST_BODY_STREAM @dfa "(?i)Content-Disposition(?:[^\r\n]*)attachment|form-data|file
    id:1 rev:1 "msg:Possible file upload" event
```

External Rules

Due to the simple rule syntax and confines of the configuration language, both basic and stream matching rules only allow for simple matching logic. Some more advanced logic can be obtained through features such as rule chaining, however when more control is required, external rules are available. External rules refer to a rule defined externally to the configuration and can thus be much more expressive. Currently the Lua scripting language is available through external rules via the `RuleExt` directive, which refers to an external lua script.

```
RuleExt lua:example.lua id:1 rev:1 phase:REQUEST_HEADER
```

```
-- example.lua
local ib = ...

-- This must be defined before assignment
-- so that the self-recursive call uses
-- the local variable instead of a global.
local printValues
local k
local v

-- Create a local function for printing values
printValues = function(name,value)
  if value then
    if type(value) == 'table' then
      -- Print the table.
      for k,v in pairs(value) do
        printValues(name.."."..k, v)
      end
    else
      ib:logInfo(name.."="..value)
    end
  end
end

-- Create a local function to fetch/print fields
local fieldPrint = function(name)
  printValues(name, ib:get(name))
end

-- Print out all the available fields
for k,v in pairs(ib:getFieldList()) do
  fieldPrint(v)
end

-- Return the result (0:FALSE 1:TRUE) to the rule engine
return 0
```

Common Rule Components

Most rules share a common set of metadata attributes as well as actions.

Metadata

Rule metadata is specified using the following modifiers:

- `id` - globally unique identifier, in the form `vendorPrefix/vendorRuleId`. It is recommended that all rule IDs within a set have at least a common prefix. Additionally, you are encouraged to further delimit by category or type. For example: `qualys/sqli/5`.

- `rev` - revision, which is used to differentiate between two versions of the same rule; it defaults to 1 if not specified.
- `msg` - message that will be used when the rule triggers. Rules that generate events must define a message.
- `tag` - assigns one or more tags to the rule; tags are used to classify rules and events (as events inherit all tags from the rule that generates them).
- `phase` - determines when the rule will run (Not available in streaming rules as these are triggered on new data)
- `severity` - determines the seriousness of the finding (0-100)
- `confidence` - determines the confidence the rule has in its logic (0-100)

Events

During a transaction, one or more events may be generated (see the `event` action). Each event has the following attributes - many of which is controlled by the rule metadata.

Event ID

Uniquely generated (for the transaction) event identifier

Event Type

Type of event. Currently this is one of:

Observation

An event which may contribute to a further decision.

Alert

An event which denotes the transaction should be logged.

Rule ID

The rule which created the event, if it was generated by a rule.

Field(s)

A optional list of inspected fields which contributed to the event.

Tag(s)

An optional list of tags used to classify the event.

Data

Arbitrary data associated with the event. This is to be treated as opaque and will be accompanied with a length in bytes.

Message

A text message associated with the event.

Confidence

A positive integer value ranging from 0-100 denoting the percent of confidence that the event is accurate.

Severity

A positive integer value ranging from 0-100 denoting the severity (weight) that this event may pose if accurate.

Recommended Action

The event creator is recommending an action to be taken. This is currently one of:

Log

Log the transaction.

Block

Block the transaction.

Ignore

Allow the transaction without further inspection.

Allow

Allow the transaction, but continue inspecting.

Suppression

Denotes the event should be suppressed and for what reason. Currently this is one of:

None

The event is not to be suppressed.

False Positive

The event was determined to be a false positive.

Replaced

The event was replace with a later event.

Incomplete

The event may contain incomplete information or be based off of incomplete information.

Other

The event was supressed for an unspecified reason.

Request and Response Body Handling

Request and response headers are generally limited in size and thus easy to handle. This is especially true in a proxy deployment, where buffering is possible. Proxies will typically cache request and response headers, making it easy to perform inspection and reliably block when necessary.

The situation is different with request and response bodies, which can be quite big. For example, request bodies may carry one or more files; response bodies too often deliver files, and some HTML responses can get quite big too. Even when sites do not normally have large request bodies, they are under the control of attackers, and they may intentionally submit large amounts of data in an effort to bypass inspection.

Let's look of what might be of interest here:

Inspection

: Do we want to inspect a particular request or response body? Whereas it would be rare not to want inspect a request body, it's quite common with response bodies, because many carry static files and images. We can decide by looking at the `Content-Type` header.

Processing

: After we decide to inspect a body, we need to determine how to process it, after which inspection can take place. It's only in the simplest case, when the body is treated as a continuous stream of bytes, is that no processing is needed. Content types such as `application/x-www-form-urlencoded` and `multipart/form-data` must be parsed before fine-grained analysis can be undertaken. In many cases we may need to process a body in more than one way to support all the desired approaches to analysis.

Buffering

: Reliable blocking is possible only when all of the data is buffered: accumulate the entire request (or response) until the inspection is complete, and then you release it all once. Blocking without buffering can be effective, but such approach is susceptible to evasion in edge cases. The comfort of reliable blocking comes at a price. End user performance may degrade, because rather than receiving data as it becomes available, the proxy must wait to receive the last byte of the data to let it through. In some cases (e.g., WebSockets) there is an expectation that chunks of data travel across the wire without delay. And, of course, buffering increases memory consumption required for inspection.

Logging

: Finally, we wish to be able to log entire transaction for post-processing or evidence. This is easy to do when all of data is buffered, but it should also be possible even when buffering is not enabled.

Request body processing

IronBee comes with built-in logic that controls the default handling of request body data. It will correctly handle `application/x-www-form-urlencoded` and `multipart/form-data` requests. Other formats will be added as needed.

Chapter 2: Server Configuration

...

Apache Trafficserver Plugin Configuration

In order to load IronBee into Apache Trafficserver (ATS) you must edit plugins.config to first load the IronBee library using the ATS loader plugin, then load the IronBee plugin with an IronBee configuration.

```
### plugins.config
# Load the IronBee library
/usr/local/ironbee/lib/libloader.so /usr/local/ironbee/lib/libironbee.so

# Load the IronBee plugin
/usr/local/ironbee/lib/ts_ironbee.so /usr/local/ironbee/etc/ironbee-ts.conf
```

TrafficServer Library Loader Plugin

Description: Load arbitrary shared libraries.

Syntax: `libloader.so` *`sharedlib.so`*

Default: `none`

Version: 0.5.0

TrafficServer IronBee Plugin

Description: Bootstrap the trafficserver ironbee plugin.

Syntax: *`/path/to/`*`ts_ironbee.so` *`[-v loglevel] [-l log-file-name] [-L] /path/to/ironbee.conf`*

Default: `none`

Version: 0.8.0

The default log filename is "ts-ironbee.log" and log level 4. The optional *-l name* sets a different name, while *-v n* sets a log level, and *-L* disables logging Ironbee messages by Trafficserver altogether (for users with an alternative logger configured).

Apache Httpd Module Configuration

In order to load IronBee into Apache httpd you must edit the httpd.conf to first load the IronBee module, then configure the module to bootstrap the IronBee library.

```
### httpd.conf

# Load the IronBee module
LoadModule ironbee_module /usr/local/ironbee/lib/mod_ironbee.so
```

```
# Bootstrap the IronBee library
<IfModule ironbee_module>
    # Specify the IronBee configuration file.
    IronbeeConfigFile /usr/local/ironbee/etc/ironbee-httpd.conf

    # Send raw headers (from client) to Ironbee for inspection
    # instead of the (potentially modified) httpd headers.
    IronbeeRawHeaders On
</IfModule>
```

From here, you can configure Apache httpd as either a webserver or a proxy server.

IronbeeConfigFile

Description: Location of the main IronBee configuration file.

Syntax: `IronbeeConfigFile` *`/path/to/ironbee.conf`*

Default: `none`

Version: 0.5.0

IronbeeRawHeaders

Description: Use the raw (client) headers or the proxy headers in ironbee.

Syntax: `IronbeeRawHeaders On|Off`

Default: `On`

Version: 0.5.0

Setting this to `On` will cause IronBee to be sent the raw headers from the client. Setting this to `Off` will cause IronBee to be sent the proxied headers (those seen by the server).

IronbeeFilterInput

Description: Filter HTTP request body data.

Syntax: `IronbeeFilterInput On|Off`

Default: `On`

Version: 0.8.0

Determines whether HTTP request bodies (where present) are sent to Ironbee for inspection.

IronbeeFilterOutput

Description: Filter HTTP response body data.

Syntax: `IronbeeFilterOutput On|Off`

Default: `On`

Version: 0.8.0

Determines whether HTTP response bodies (where present) are sent to Ironbee for inspection.

IronbeeLog

Description: Enable/Disable apache logging.

Syntax: `IronbeeLog On|Off`

Default: `On`

Version: 0.8.0

Determines whether Ironbee messages are logged to the HTTPD error log. If disabled, some other logger should be configured.

IronbeeLogLevel

Description: Set default Ironbee log level.

Syntax: `IronbeeLogLevel 0-9`

Default: `4`

Version: 0.8.0

Sets the initial log level for Ironbee messages (ignored if IronbeeLog is Off).

Nginx Configuration

In configure IronBee in nginx you must edit the nginx.conf to bootstrap the IronBee library.

```
### nginx.conf

...

http {
    # Bootstrap the IronBee library
    ironbee_config_file /usr/local/ironbee/etc/ironbee-httpd.conf;

    # Setup logging
    ironbee_logger On;
    ironbee_loglevel 4;

    ...
}
```

From here, you can configure nginx.

ironbee_config_file

Description: Location of the main IronBee configuration file.

Syntax: `ironbee_config_file` *`/path/to/ironbee.conf`*`;`

Default: `none`

Version: 0.7.0

This needs to go in the nginx "http" block.

ironbee_logger

Description: Enable the ironbee logger.

Syntax: `ironbee_logger "On"|"Off";`

Default: `none`

Version: 0.7.0

This needs to go in the nginx "http" block.

ironbee_loglevel

Description: Set the (numeric) ironbee log level.

Syntax: `ironbee_loglevel` *`numeric-level_0-10`*`;`

Default: `none`

Version: 0.7.0

This needs to go in the nginx "http" block.

Chapter 3: IronBee Configuration

The IronBee configuration is loaded from the server container. The syntax is similar to the Apache httpd server configuration. The following rules apply:

- Escape sequences are as in JavaScript (section 7.8.4 in ECMA-262), except within PCRE regular expression patterns, where PCRE escaping is used
- Lines that begin with # are comments
- Lines are continued on the next line when \ is the last character on a line

The IronBee configuration defines general configuration as well as site and location mappings, which can each have their own configuration.

```
# Main Configuration
SensorId 13AABA8F-2575-4F93-83BF-C87C1E8EECCE
...

# Site1
<Site site1>
    SiteId 0B781B90-CE3B-470C-952C-5F2878EFFC05
    Hostname site1.example.com
    Service 10.0.1.100:80

    ...

    <Location /directory1>
        ...
    </Location>
</Site>

# Site2
<Site site2>
    SiteId 8B3BA3DE-2727-4737-9230-4A1D110E6C87
    Hostname site2.example.com
    Service 10.0.5.100:80

    ...
</Site>

# Default Site
<Site default>
    SiteId F89E43B3-EB96-44F0-BE1C-B4673B96DF9C
    Hostname *
    Service *:*

    ...
</Site>
```

The following is a reference for all IronBee directives where the context refers to the possible locations within the configuration file.

AuditEngine

Description: Configures the audit log engine.

Syntax: `AuditEngine On|Off|RelevantOnly`

Default: `RelevantOnly`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.3

Setting `AuditEngine` to `RelevantOnly`, the default, does not log any transactions in itself. Instead, further activity (e.g., a rule match) is required for a transaction to be recorded. Setting `AuditEngine` to `On` activates audit logging for **all transactions**, which may cause a large amount of data to be logged.

AuditLogBaseDir

Description: Configures the directory where individual audit log entries will be stored. This also serves as the base directory for `AuditLogIndex` if it uses a relative path.

Syntax: `AuditLogBaseDir` *`directory`*

Default: `"/var/log/ironbee"`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.3

AuditLogDirMode

Description: Configures the directory mode that will be used for new directories created during audit logging.

Syntax: `AuditLogDirMode` *`octal-mode`*

Default: `0700`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

AuditLogFileMode

Description: Configures the file mode that will be used when creating individual audit log files.

Syntax: `AuditLogFileMode` *`octal-mode`*

Default: `0600`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.6

AuditLogIndex

Description: Configures the location of the audit log index file.

Syntax: `AuditLogIndex None|`*`filename`*

Default: `ironbee-index.log`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

Relative filenames are based off the `AuditLogBaseDir` directory and specifying `None` disables the index file entirely.

AuditLogIndexFormat

Description: Configures the format of the entries logged in the audit log index file.

Syntax: `AuditLogIndexFormat` *`format-string`*

Default: `"%T %h %a %S %s %t %f"`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

- **%%** The percent sign
- **%a** Remote IP-address
- **%A** Local IP-address
- **%h** HTTP Hostname
- **%s** Site ID
- **%S** Sensor ID
- **%t** Transaction ID

- **%T** Transaction timestamp (YYYY-MM-DDTHH:MM:SS.ssss+/-ZZZZ)
- **%f** Audit log filename (relative to `AuditLogBaseDir`)

AuditLogParts

Description: Configures which parts will be logged to the audit log.

Syntax: `AuditLogPart [+|-]partType ...`

Default: `default`

Context: Any

Cardinality: 0..n

Module: core

Version: 0.4

An audit log consist of many parts; `AuditLogParts` determines which parts are recorded by default. The parts are inherited into child contexts (Site, Location, etc). Specifying a part with +/- operator will add or remove the given part from the current set of parts. Specifying the first option without +/- operators will cause all options to be overridden and the list of options will be the only options set. Here is what your configuration might look like:

```
AuditLogParts minimal +request -requestBody +response -responseBody
```

The above first resets the list of parts to **minimal**, adds all the **request** parts except the **requestBody**, then adds all the **response** parts except the **responseBody**.

Later, in a sub-context, you may wish to enable response body logging and thus can just specify this part with the + operator:

```
<Location /some/path>
        AuditLogParts +responseBody
    </Location>
```

If you already had response body logging enabled, but didn't want it any more, you would write:

```
<Location /some/path>
        AuditLogParts -responseBody
    </Location>
```

Audit Log Part Names:

- **header:** Audit Log header (required)
- **events:** List of events that triggered
- **requestMetadata:** Information about the request
- **requestHeaders:** Raw request headers
- **requestBody:** Raw request body

- **requestTrailers:** Raw request trailers
- **responseMetadata:** Information about the response
- **responseHeaders:** Raw response headers
- **responseBody:** Raw response body
- **responseTrailers:** Raw response trailers
- **debugFields:** Currently not implemented

Audit Log Part Group Names:

These are just aliases for multiple parts.

- **none:** Removes all parts
- **minimal:** Minimal parts (currently **header** and **events** parts)
- **default:** Default parts (currently **minimal** and request/response parts without bodies)
- **request:** All request related parts
- **response:** All response related parts
- **debug:** All debug related parts
- **all:** All parts

AuditLogSubDirFormat

Description: Configures the directory structure created under the *AuditLogBaseDir* directory. This is a *strftime(3)* format string allowing the directory structure to be created based on date/time.

Syntax: `AuditLogSubDirFormat` *`format-string`*

Default: `403`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

DefaultBlockStatus

Description: Configures the default HTTP status code used for blocking.

Syntax: `DefaultBlockStatus` *`http-status-code`*

Default: `403`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

GeoIPDatabaseFile

Description: Configures the location of the geoip database file.

Syntax: `GeoIPDatabaseFile` *`filename`*

Default: `/usr/share/geoip/GeoLiteCity.dat`

Context: Any

Cardinality: 0..1

Module: geoip

Version: 0.4

Hostname

Description: Maps hostnames to a Site.

Syntax: `Hostname` *`hostname`*

Default: `*` (any)

Context: Site

Cardinality: 0..n

Module: core

Version: 0.4

The `Hostname` directive establishes a mapping between a Site and one or more hostnames. To map IP/Port pairs to a Site, see the `Service` directive.

In the simplest case, a site will occupy a single hostname:

```
Hostname www.ironbee.com
```

More often than not, however, several names will be used:

```
Hostname www.ironbee.com
Hostname ironbee.com
```

Wildcards are permitted when there are multiple names under a common domain. Only one wildcard character per hostname is allowed and it must currently be on the left-hand side:

```
Hostname ironbee.com
Hostname *.ironbee.com
```

Finally, to match any hostname (which you will need to do in default sites), use a single asterisk, which is the default if no `Hostname` directive is specified for a site:

```
Hostname *
```

Include

Description: Includes external file into configuration.

Syntax: `Include` *`filename`*

Default: None

Context: Any

Cardinality: 0..n

Module: core

Version: 0.5

Allows inclusion of another file into the current configuration file. The following line will include the contents of the file `sites.conf` into configuration:

```
Include conf/sites.conf
```

The file must exist and be accessible or an error is generated (use `IncludeIfExists` if this is not the case). If you specify a relative path, the location of the configuration file containing this directive will be used to resolve it.

IncludeIfExists

Description: Includes external file into configuration if it exists and is accessible.

Syntax: `IncludeIfExists` *`filename`*

Default: None

Context: Any

Cardinality: 0..n

Module: core

Version: 0.7

As `Include`, but allows for optional inclusion without causing a configuration error if the file does not exist (as would the `Include` directive).

InitCollection

Description: Initializes a locally scoped collection data field for later use and optional persistence.

Syntax: `InitCollection` *`collection-name uri [param1 param2 ... paramN]`*

Default: None

Context: Any

Cardinality: 0..1

Module: core, persist

Version: 0.7

Initializes a collection from the initializer. The initializer format depends on the implementation. There are multiple URI formats supported, which are described below.

Core Functionality

`vars: key1=val1 key2=val2 ... keyN=valN`

The `vars` URI allows initializing a collection of simple key/value pairs.

```
InitCollection MY_VARS vars: key1=value1 key2=value2
```

`json-file:///path/file.json [persist]`

The json-file URI allows loading a more complex collection from a JSON formatted file. If the optional persist parameter is specified, then anything changed is persisted back to the file at the end of the transaction. Next time the collection is initialized, it will be from the persisted data.

```
InitCollection MY_JSON_COLLECTION json-file:///tmp/ironbee/persist/test1.json
InitCollection MY_PERSISTED_JSON_COLLECTION json-file:///tmp/ironbee/persist/test2.json persist
```

Persist Module

The persist module allows for some more advanced persistence, such as providing multiple instances of persisted collection as well as expiration. To load this functionality you must load the persist module separately.

`persist-fs:///path/to/persisted/data key=VALUE [expire=SECONDS]`

The `persist-fs` URI allows specifying a path to store persisted data. The `key` parameter specifies a value to identify an instance of the collection. The `key` value can be any text or a field expansion (e.g., `%{MY_VAR_NAME}`). The `expire` parameter allows setting the expiration of the data stored in the collection in seconds. On initialization, the collection is populated from the persisted data. If the data is expired when the collection is initialized, it is discarded and an empty collection will be created.

```
LoadModule ibmod_persist.so

...

# Initialize a collection from the persistence store keyed off of REMOTE_ADDR.
# The IP collection is now associated with the REMOTE_ADDR and any updates
# will be persisted back to the persistence store with the REMOTE_ADDR key.
# Different instances of the IP collection are stored based on the key. The
# data stored in this collection will expire 300 seconds after persisted.
InitCollection IP persist-fs:///tmp/ironbee/persist key=%{REMOTE_ADDR} expire=300

# Check a value from the persisted collection to determine if a block should
# occur.
Rule IP:block @gt 0 id:persist/isblocked phase:REQUEST_HEADER event block:immediate
```

```
# Perform some checks, setting block flag.
Rule ... block

# Update the persistent IP collection. This will store a block=1 parameter
# for the IP collection associated with the REMOTE_ADDR key. If the IP collection
# is pulled from the store again (within the expiration), then the rule above
# will immediatly block the transaction.
Rule FLAGS:block.count() @gt 0 id:persist/setblock phase:REQUEST setvar:IP:block=1

# After the transaction completes, the modified values are persisted and the
# persisted IP:block=1 will be used to block all transactions from the same IP
# address for the next 300 seconds.
```

InitCollectionIndexed

Description: As InitCollection but also index key.

Syntax: `InitCollectionIndexed` *`collection-name uri [param1 param2 ... paramN]`*

Default: None

Context: Any

Cardinality: 0..1

Module: core, persist

Version: 0.8

InitVar

Description: Initializes a locally scoped variable data field for later use.

Syntax: `InitVar` *`variable-name initial-value`*

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: 0.6

InitVarIndexed

Description: As InitVar but also index key.

Syntax: `InitVarIndexed` *`variable-name initial-value`*

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: 0.8

InspectionEngineOptions

Description: Configures options for the inspection engine.

Syntax: `InspectionEngineOptions [+|-]option ...`

Default: `default`

Context: Any

Cardinality: 0..n

Module: core

Version: 0.7

The inspection engine allows setting options; `InspectionEngineOptions` controls these options. The options are inherited into child contexts (Site, Location, etc). Specifying an option with +/- operator will add or remove the given option from the current set. Specifying the first option without +/- operators will cause all options to be overridden and the list of options will be the only options set. Here is what your configuration might look like:

```
InspectionEngineOptions all -response
```

The above first resets the inspection to **all**, then removes the **response** from being inspected.

Later, in a sub-context, you may wish to enable response response inspection and thus can just specify this part with the + operator:

```
<Location /some/path>
    InspectionEngineOptions +response
</Location>
```

If you already had response enabled, but didn't want it any more, you would write:

```
<Location /some/other/path>
    InspectionEngineOptions -response
</Location>
```

Inspection Engine Options:

- **requestHeader:** Inspect the HTTP request header (default)
- **requestBody:** Inspect the HTTP request body
- **responseHeader:** Inspect the HTTP response header
- **responseBody:** Inspect the HTTP response body

Inspection Engine Option Group Names:

These are just aliases for multiple options.

- **none:** Removes all options
- **default:** Default options (currently request header only)

- **request:** All request related options
- **response:** All response related options
- **all:** All options

LoadEudoxus

Description: Loads an external Eudoxus Automata into IronBee.

Syntax: `LoadEudoxus` *`name file`*

Default: None

Context: Main

Cardinality: 0..n

Module: ee

Version: 0.7

This directive will load an external eudoxus automata from `file` into the engine with the given `name`. Once loaded, the automata can then be used with the associated eudoxus rule operators such as the `ee_match_any` operator.

The eudoxus automata is a precompiled and optimized automata generated by the ac_generator and ec commands in the `automata/bin` directory. Currently, as of IronBee 0.7, a modified Aho-Corasick algorithm is implemented which can handle very large external dictionaries. Refer to the IronAutomata Documentation [https://www.ironbee.com/docs/devexternal/ironautomata.html] for more information.

LoadModule

Description: Loads an external module into configuration.

Syntax: `LoadModule` *`module`*

Default: None

Context: Main

Cardinality: 0..n

Module: core

Version: 0.4

This directive will add an external module to the engine, potentially making new directives available to the configuration.

Location

Description: Creates a subcontext that can have a different configuration.

Syntax: `<Location` *`path`*`>...</Location>`

Default: None

Context: Site

Cardinality: 0..n

Module: core

Version: 0.4

A sub-context created by this directive initially has identical configuration to that of the site it belongs to. Further directives are required to introduce changes. Locations are evaluated in the order in which they appear in the configuration file. The first location that matches request path will be used. This means that you should put the most-specific location first, followed by the less specific ones.

```
Include rules.conf

    <Site site1>
        Service *:80
        Service 10.0.1.2:443
        Hostname site1.example.com

        <Location /prefix/app1>
            RuleEnable all
        </Location>

        <Location /prefix>
            RuleEnable tag:GenericRules
        </Location>
    </Site>
```

Log

Description: Configures the location of the log file.

Syntax: `Log default|`*`filename`*

Default: `default`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

LogHandler

Description: Configures the log handler.

Syntax: `LogHandler` *`name`*

Default: `None`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.3

DEPRECATED - Do not use. The log handler is now automatically set by the servers.

> **Note**
>
> The log handler allows the log to be handled by another facility (currently the server). For Apache Traffic Server, this should be set to `"ironbee-ts"` and for Apache Web Server, this should be set to `"mod_ironbee"`. Using the log handler overrides the `Log` directive.

LogLevel

Description: Configures the detail level of the entries recorded to the log.

Syntax: `LogLevel` *`level`*

Default: `4`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.4

The following log levels are supported (either numeric or text):

- `0 - emergency` - system unusable
- `1 - alert` - crisis happened
- `2 - critical` - crisis coming
- `3 - error` - error occurred
- `4 - warning` - error likely to occur
- `5 - notice` - something unusual happened
- `6 - info` - informational messages
- `7 - debug` - debugging: transaction state changes
- `8 - debug2` - debugging: log of activities carried out
- `9 - debug3` - debugging: activities, with more detail
- `10 - trace` - debugging: developer log messages

LuaCommitRules

Description: Signal the lua module to add (commit) all lua defined rules to the rule engine. Note this is a temporary fix and will be removed in a later version.

Syntax: `LuaCommitRules`

Default: None

Context: Main

Cardinality: 0..1
Module: lua
Version: 0.7
DEPRECATED: Do not use after IronBee 0.7
Example:

```
LuaInclude "rules1.lua"
LuaInclude "rules2.lua"
LuaInclude "rules3.lua"
 ...
LuaCommitRules
```

LuaLoadModule

Description: Load a Lua module (similar to LoadModule).
Syntax: `LuaLoadModule` *`lua-file`*
Default: None
Context: Main
Cardinality: 0..1
Module: lua
Version: 0.7
Example:

```
LuaLoadModule "threat_level.lua"
```

LuaInclude

Description: Execute a Lua script as a configuration file.
Syntax: `LuaInclude` *`lua-file`*
Default: None
Context: Main
Cardinality: 0..1
Module: lua
Version: 0.7
Example:

```
LuaInclude "rules.lua"
```

ModuleBasePath

Description: Configures the base path where IronBee modules are loaded.

Syntax: `ModuleBasePath` *`path`*

Default: The `lib` directory under the IronBee install prefix.

Context: Main

Cardinality: 0..1

Module: core

Version: 0.4

PcreMatchLimit

Description: Configures the PCRE library match limit.

Syntax: `PcreMatchLimit` *`limit`*

Default: 5000

Context: Main

Cardinality: 0..1

Module: pcre

Version: 0.4

From the `pcreapi` manual: “The match_limit field provides a means of preventing PCRE from using up a vast amount of resources when running patterns that are not going to match, but which have a very large number of possibilities in their search trees. The classic example is a pattern that uses nested unlimited repeats.”

PcreMatchLimitRecursion

Description: Configures the PCRE library match limit recursion.

Syntax: `PcreMatchLimitRecursion` *`limit`*

Default: 5000

Context: Main

Cardinality: 0..1

Module: pcre

Version: 0.4

From the `pcreapi` manual: “The match_limit_recursion field is similar to match_limit, but instead of limiting the total number of times that match() is called, it limits the depth of recursion. The recursion depth is a smaller number than the total number of calls, because not all calls to match() are recursive. This limit is of use only if it is set smaller than match_limit.”

RequestBuffering

Description: Enable/disable request buffering.

Syntax: `RequestBuffering On|Off`

Default: `Off`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.6

Control request buffering - holding the request during inspection. Currently the HTTP header is always buffered, but this must be enabled for the request body to be buffered.

> **Note**
>
> This may be renamed to `RequestBodyBuffering` in a future release.

RequestBodyBufferLimit

Description: Configures the size of the request body buffer.

Syntax: `RequestBodyBufferLimit` *`byte_limit`*

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: Not implemented yet

RequestBodyBufferLimitAction

Description: Configures what happens when the buffer is smaller than the request body.

Syntax: `RequestBodyBufferLimitAction Reject|RollOver`

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: Not implemented yet

When `Reject` is configured, the transaction with a body larger than the buffer will be blocked. With `RollOver` selected, the buffer will be used to keep as much data as possible, but any overflowing data will be allowed to the backend. Request headers will be sent before the first overflow batch. In detection-only mode, `Reject` is converted to `RollOver`.

ResponseBuffering

Description: Enable/disable response buffering.

Syntax: `ResponseBuffering On|Off`

Default: `Off`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.6

Control response buffering - holding the response during inspection. Currently the HTTP header is always buffered, but this must be enabled for the response body to be buffered.

> **Note**
>
> This may be renamed to `ResponseBodyBuffering` in a future release.

ResponseBodyBufferLimit

Description: Configures the size of the response body buffer.

Syntax: `ResponseBodyBufferLimit` *`byte_limit`*

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: Not implemented yet

ResponseBodyBufferLimitAction

Description: Configures what happens when the buffer is smaller than the response body.

Syntax: `ResponseBodyBufferLimitAction Reject|RollOver`

Default: None

Context: Any

Cardinality: 0..1

Module: core

Version: Not implemented yet

When `Reject` is configured, the transaction with a body larger than the buffer will be blocked. With `RollOver` selected, the buffer will be used to keep as much data as possible, but any overflowing data will be allowed to the client. Response headers will be sent before the first overflow batch. In detection-only mode, `Reject` is converted to `RollOver`.

Rule

Description: Loads a rule and, in most contexts, enable the rule for execution in that context.

Syntax: `Rule` *`input [input2 ... inputN] @operator op_param [modifiers]`*

Default: None

Context: Any

Cardinality: 0..n

Module: rules

Version: 0.4

> **Note**
>
> Loading a rule will, in most contexts, also enable the rule to be executed in that context. However, the main configuration context is special. Loading a rule in the main configuration context will *NOT* enable the rule, but just load it into memory so that it can be shared by other contexts. You must explicitly use `RuleEnable` in another context to enable the rule.

RuleBasePath

Description: Configures the base path where external IronBee rules are loaded.

Syntax: `RuleBasePath` *`path`*

Default: The `lib` directory under the IronBee install prefix.

Context: Main

Cardinality: 0..1

Module: core

Version: 0.4

RuleDisable

Description: Disables a rule from executing in the current configuration context.

Syntax: `RuleDisable "all" | "id:"`*`id`* `| "tag":`*`name`* `...`

Default: None

Context: Any

Cardinality: 0..n

Module: rules

Version: 0.4

Rules can be disabled by id or tag. Any number of id or tag modifiers can be specified per directive. All disables are processed after enables. See the `RuleEnable` directive for an example.

RuleEnable

Description: Enables a rule for execution in the current configuration context.

Syntax: `RuleEnable "all" | "id:"`*`id`* `| "tag":`*`name`* `...`

Default: None

Context: Any

Cardinality: 0..n

Module: rules

Version: 0.4

Rules can be disabled by id or tag. Any number of id or tag modifiers can be specified per directive. All enables are processed before disables. For example:

```
Include "rules/big_ruleset.conf"

    <Site foo>
        Hostname foo.example.com
        RuleEnable id:1234
        RuleEnable id:3456 tag:SQLi
        RuleDisable id:5678 tag:experimental tag:heavyweight
    </Site>
```

RuleEngineLogData

Description: Configures the data logged by the rule engine.

Syntax: `RuleEngineLogData [+|-]option ...`

Default: `ruleExec`

Context: Any

Cardinality: 0..n

Module: core

Version: 0.6

The following data type options are supported:

- `tx` - Log the transaction:

```
TX_START clientip:port site-hostname
    ...
TX_END
```

- `requestLine` - Log the HTTP request line:

```
REQ_LINE method uri version-if-given
```

- `requestHeader` - Log the HTTP request header:

```
REQ_HEADER name: value
```

- `requestBody` - Log the HTTP request body, possibly in multiple chunks:

```
REQ_BODY size data
```

- `responseLine` - Log the HTTP response line:

```
RES_LINE version status message
```

- `responseHeader` - Log the HTTP response header:

```
RES_HEADER name: value
```

- `responseBody` - Log the HTTP response body, possibly in multiple chunks:

```
RES_BODY size data
```

- `phase` - Log the phase about to execute:

```
PHASE name
```

- `rule` - Log the rule executing:

```
RULE_START rule-type
    ...
RULE_END
```

- `target` - Log the target being inspected:

```
TARGET full-target-name {NOT_FOUND|field-type field-name field-value}
```

- `transformation` - Log the transformation being executed:

```
TFN tfn-name(param) {ERROR error}
```

- `operator` - Log the operator being executed:

```
OP op-name(param) TRUE|FALSE {ERROR error}
```

- `action` - Log the action being executed:

```
ACTION action-name(param) {ERROR error}
```

- `event` - Log the event being logged:

```
EVENT rule-id type action [confidence/severity] [csv-tags] msg
```

- `audit` - Log the audit log filename being written:

```
AUDIT audit-log-filename
```

The following alias options are supported:

- `request` - Alias for: `requestLine`, `requestHeader`, `requestBody`

- `response` - Alias for: `responseLine`, `responseHeader`, `responseBody`
- `ruleExec` - Alias for: `phase`, `rule`, `target`, `transformation`, `operator`, `action`, `actionableRulesOnly`
- `default` - Alias for: `ruleExec`
- `all` - Alias for all data options

The following filter options are supported:

- `actionableRulesOnly` - Filter option indicating that only rules that were actionable (actions executed) are logged - any rule specific logging are delayed/suppressed until at least one action is executed.

RuleEngineLogLevel

Description: Configures the logging level which the rule engine will write logs.

Syntax: `RuleEngineLogLevel` *`level`*

Default: `info`

Context: Any

Cardinality: 0..1

Module: core

Version: 0.6

RuleExt

Description: Creates a rule implemented externally, either by loading the rule directly from a file, or referencing a rule that was previously declared by a module.

Syntax: `RuleExt` *`ruleLocation actions`*

Default: None

Context: Site, Location

Cardinality: 0..n

Module: rules

Version: 0.4

To load a Lua rule:

```
RuleExt lua:/path/to/rule.lua phase:REQUEST
```

RuleMarker

Description: Creates a rule marker (placeholder) which will not be executed, but instead should be overridden. The idea is that rule sets can include placeholders for optional custom rules which can be overridden, but still allow the rule set writer to maintain execution order.

Syntax: `RuleMarker id:`*`id`* `phase:`*`phase`*

Default: None

Context: Any

Cardinality: 0..n

Module: rules

Version: 0.5

To mark and later replace a rule:

```
Rule ARGS @rx foo id:1 rev:1 phase:REQUEST

# Allow the administrator to set MY_VALUE in another context
RuleMarker id:2 phase:REQUEST

Rule MY_VALUE @gt 0 id:3 rev:1 phase:REQUEST setRequestHeader:X-Foo:%{MY_VALUE}

<Site test>
    Hostname *

    Rule &ARGS @gt 5 id:2 phase:REQUEST setvar:MY_VALUE=5
    RuleEnable all
</Site>
```

In the above example, rule id:2 in the main context would be replaced by the rule id:2 in the site context, then the rules would execute id:1, id:2 and id:3. If Rule id:2 was not replaced in the site context, then rules would execute id:1 then id:3 as id:2 is only a marker (placeholder).

SensorId

Description: Unique sensor identifier.

Syntax: `SensorId` *`sensor_id`*

Default: None

Context: Main

Cardinality: 0..1

Module: core

Version: 0.4

TODO: Can we make this directive so that, if not defined, we attempt to detect server hostname and use that as ID?

Service

Description: Maps IP and Port to a site.

Syntax: `Service` *`ip`*`:`*`port`*

Default: `*:*` (any)

Context: Site

Cardinality: 0..n

Module: core

Version: 0.6

The `Service` directive establishes a mapping between a Site and one or IP/Port pairs. To map hostnames to a Site, see the `Hostname` directive.

In the simplest case, a site will occupy a single IP/Port pair:

```
Service 192.168.32.5:80
```

More often than not, however, several mappings will be used:

```
Service 192.168.32.5:80
Service 192.168.32.6:443
```

Wildcards are permitted for both IP and Port:

```
Service *:80
Service 192.168.32.5:*
```

To match any IP address on any Port (which you will need to do in default sites), use wildcards for both IP and Port, which is the default if no `Service` directive is specified for a site:

```
Service *:*
```

Site

Description: A site is one of the main concepts in the configuration in IronBee. The idea is to have an element to correspond to real-life web sites. With most web sites there is an one-to-one mapping to domain names, but our mapping mechanism is quite flexible: you can have one site per domain name, many domain names for a single site, or even have one domain name shared among several sites.

Syntax: `<Site site_name>...</Site>`

Default: None

Context: Main

Cardinality: 0..n

Module: core

Version: 0.1

At the highest level, a configuration will contain one or more sites. For example:

```
<Site site1>
    Service *:80
    Hostname site1.example.com
```

```
    Hostname site1-alternate.example.com
</Site>

<Site site2>
    Service *:80
    Service 10.0.1.2:443
    Hostname site2.example.com
</Site>

<Site default>
    Service *:*
    Hostname *
</Site>
```

Before it can process a transaction, IronBee will examine the current configuration looking for a site to assign the transaction. Sites are processed in the configured order where the first matching site is chosen. A default site can be specified as the last site using wildcards when all previous sites fail to match. The `Site` directive only establishes configuration boundaries and assigns a unique handle to each site; the `Service` and `Hostname` directives are responsible for the mapping.

SiteId

Description: Unique site identifier.

Syntax: `SiteId` *`site_id`*

Default: None

Context: Site

Cardinality: 0..1

Module: core

Version: 0.4

TODO: Can we make this directive so that, if not defined, we attempt to detect site hostname and use that as ID?

StreamInspect

Description: Creates a streaming inspection rule, which inspects data as it becomes available, outside rule phases.

Syntax: `StreamInspect` *`stream-field`* `"@"`*`operator op_param modifiers`*

Context: Site, Location

Cardinality: 0..n

Module: rules

Version: 0.4

Normally, rules run in one of the available phases, which happen at strategic points in transaction lifecycle. Phase rules are convenient to write, because all the relevant data is available for inspection. However, there are situations when it is not possible to have access to all of the data

in a phase. This is the case, for example, when a request body is very large, or when buffering is not allowed.

Streaming rules are designed to operate in these circumstances. They are able to inspect data as it becomes available, be it a dozen of bytes, or a single byte.

The syntax of the `Inspect` directive is similar to that of `Rule`, but there are several restrictions:

- Only one input can be used. This is because streaming rules attach to a single data source.
- The `phase` modifier cannot be used, as streaming rules operate outside of phases.
- Only `REQUEST_BODY_STREAM` and `RESPONSE_BODY_STREAM` can be used as inputs.
- Only the `pm`, and `dfa` operators can be used.
- Transformation functions are not yet supported.

TxDump

Description: Diagnostics directive to dump (log) transaction data for debugging purposes.

Syntax: `TxDump` *`event destination [data]`*

Default: `None`

Context: Any

Cardinality: 0..1

Module: devel

Version: 0.7

The *`event`* field allows indicating *when* you want the data to be written and is one of:

- `TxStarted` - Transaction started.
- `TxProcess` - Transaction processing (between request and response).
- `TxContext` - Transaction configuration context chosen.
- `RequestStart` - Request started.
- `RequestHeader` - Request headers have been processed.
- `Request` - Full request has been processed.
- `ResponseStart` - Response started.
- `ResponseHeader` - Response headers have been processed.
- `Response` - Full response has been processed.
- `TxFinished` - Transaction is finished.
- `Logging` - Logging phase.
- `PostProcess` - Post-processing phase.

The *`destination`* field allows specifying *where* you want to write the data and is one of the following:

- `stderr` - Write to standard error.

- `stdout` - Write to standard output.
- `ib` - Write to the IronBee log file.
- `file://path[+]` - Write to an arbitrary file, optionally appending to the file if the last character is a `+` character.

The *`data`* field is optional and allows specifying *what* is to be written. This can be prefixed with a `+` or a `-` character to enable or disable the data.

- `Basic` - Basic TX data.
- `Context` - Configuration context data.
- `Connection` - Connection data.
- `ReqLine` - HTTP request line.
- `ReqHdr` - HTTP request header.
- `RspLine` - HTTP response line.
- `RspHdr` - HTTP response header.
- `Flags` - Transaction flags.
- `Args` - Request arguments.
- `Data` - Transaction data.
- `Default` - Default is "Basic ReqLine RspLine".
- `Headers` - All HTTP headers.
- `All` - All data.

Examples:

```
TxDump TxContext ib Basic +Context
TxDump PostProcess file:///tmp/tx.txt All
TxDump Logging file:///var/log/ib/all.txt+ All
TxDump PostProcess StdOut All
```

Chapter 4: IronBee Diagnostics and Developer Tools

IronBee comes with a number of tools for diagnostics and development of both modules and rules.

Rule Engine Logging and Diagnostics

IronBee has a separate logging facility for rule engine which is meant to make rule diagnostics easier. Rule engine logging is controlled through two directives. There is also a variable that can be set to enable internal tracing of the rule execution engine, which is generally only useful for developers.

Typically rule execution logging is enabled as follows:

```
### Rule Engine Logging
# Control what data is logged
RuleEngineLogData default
# Control at which log level data is logged
RuleEngineLogLevel info
```

Full logging, which will generate a lot of data, can be enabled by setting `RuleEngineLogData` to `all`, however there is quite a bit of control over what is logged. See the `RuleEngineLogData` in the Chapter 3 documentation for details. An example of full rule engine logging is below.

```
### Rule Engine Logging
RuleEngineLogData all
RuleEngineLogLevel info

### Match "dirty" or "attack" in the request
Rule REQUEST_URI REQUET_HEADERS ARGS @rx "dirty|attack" \
    id:test/1 phase:REQUEST \
    severity:60 confidence:50 \
    "msg:Matched %{CAPTURE:0}" \
    capture \
    event block

### Perform an actual block action at the end of the phase
### if marked to block
Rule FLAGS:block.count() @gt 0 \
    id:block/1 phase:REQUEST \
    "msg:Blocking request" \
    status:403 \
    block:phase
```

The above will produce rule engine logging similar to below.

```
[] PHASE REQUEST_HEADER_STREAM
```

```
[] TX_START 1.2.3.4:52980 waf-test.example.com
[] REQ_LINE GET /?a=1&b=dirty&c=attack&d=4 HTTP/1.1
[] REQ_HEADER Host: waf-test.example.com
[] REQ_HEADER User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:18.0) Gecko/20100101 Firef
[] REQ_HEADER Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
[] REQ_HEADER Accept-Language: en-US,en;q=0.5
[] REQ_HEADER Connection: keep-alive
[] PHASE REQUEST_HEADER
[] PHASE REQUEST_BODY
[rule:"site/.../test/1" rev:1] RULE_START PHASE
[rule:"site/.../test/1" rev:1] TARGET "REQUEST_URI" BYTESTR "request_uri" "\/?a=1&b=dirty&c=attack&d
[rule:"site/.../test/1" rev:1] ACTION event("")
[rule:"site/.../test/1" rev:1] ACTION block("")
[rule:"site/.../test/1" rev:1] EVENT site/.../test/1 Observation Block [50/60] [] "Matched dirty"
[rule:"site/.../test/1" rev:1] TARGET REQUET_HEADERS NOT_FOUND[rule:"site/.../test/1" rev:1] TARGET
[rule:"site/.../test/1" rev:1] OP rx("dirty|attack") FALSE
[rule:"site/.../test/1" rev:1] TARGET "ARGS" BYTESTR "ARGS:b" "dirty"
[rule:"site/.../test/1" rev:1] OP rx("dirty|attack") TRUE
[rule:"site/.../test/1" rev:1] ACTION event("")
[rule:"site/.../test/1" rev:1] ACTION block("")
[rule:"site/.../test/1" rev:1] EVENT site/.../test/1 Observation Block [50/60] [] "Matched dirty"
[rule:"site/.../test/1" rev:1] TARGET "ARGS" BYTESTR "ARGS:c" "attack"
[rule:"site/.../test/1" rev:1] OP rx("dirty|attack") TRUE[rule:"site/.../test/1" rev:1] ACTION event
[rule:"site/.../test/1" rev:1] ACTION block("")
[rule:"site/.../test/1" rev:1] EVENT site/.../test/1 Observation Block [50/60] [] "Matched attack"
[rule:"site/.../test/1" rev:1] TARGET "ARGS" BYTESTR "ARGS:d" "4"
[rule:"site/.../test/1" rev:1] OP rx("dirty|attack") FALSE
[rule:"site/.../test/1" rev:1] RULE_END[rule:"site/.../block/1" rev:1] RULE_START PHASE
[rule:"site/.../block/1" rev:1] TFN count() LIST "FLAGS" 1
[rule:"site/.../block/1" rev:1] TARGET "FLAGS:block.count()" NUM "FLAGS:FLAGS" 1
[rule:"site/.../block/1" rev:1] OP gt(0) TRUE
[rule:"site/.../block/1" rev:1] ACTION status(403)
[rule:"site/.../block/1" rev:1] ACTION block("phase")
[rule:"site/.../block/1" rev:1] RULE_END
[] PHASE RESPONSE_HEADER_STREAM
[] RES_LINE HTTP/1.1 403 Forbidden
[] RES_HEADER Date: Fri, 22 Mar 2013 15:36:27 GMT
[] RES_HEADER Connection: close
[] RES_HEADER Server: ATS/3.2.2
[] PHASE RESPONSE_HEADER
[] PHASE RESPONSE_BODY_STREAM
[] PHASE POST_PROCESS
[] PHASE LOGGING
[] AUDIT /var/log/ironbee/events/118d9ea6-933d-400e-b980-cdad773dceee_9e8d34a4-1431-4a90-a79a-de9fe8
[] TX_END
```

For a production system, something like this may be suitable, which will only log events that are generated and audit log files that are written:

```
### Log only events and audit log information
```

```
RuleEngineLogData event audit
RuleEngineLogLevel notice
```

If you need further tracing through rule execution, then you can set the rule engine debug log level. This will potentially log a lot of data, but will allow for debugging issues that may not be exposed by the normal rule engine logging facility. The following will enable full trace mode in the rule engine:

```
Set RuleEngineDebugLogLevel trace
```

Command Line Tool (clipp)

IronBee includes a full featured command line tool named `clipp` (pronounced clippy) that can be used to run transactions through IronBee from various data sources such as its own native protobuf format, raw HTTP files, pcap network captures, audit logs and others. Clipp can also translate between the various formats. The interface to clipp was inspired by the brilliant socat utility (http://www.dest-unreach.org/socat/). Clipp is best explained through examples, but in general, clipp take one or more inputs, optional modifiers and a single consumer. The input data passes through modifiers into the consumer.

The following will take `raw` HTTP input files (`request.http` and `response.http`), instantiate an `ironbee` consumer with the specified configuration (`ironbee.conf`) and then push the data through the IronBee engine in the same way IronBee embedded in a webserver or proxy. Using clipp, you can test IronBee configuration files prior to moving them to a webserver or proxy. With clipp, if you specified a LogFile in the configuration, then the logs will go there, otherwise it will go to stderr.

```
$ clipp raw:request.http,response.http ironbee:ironbee.conf
```

Multiple transactions are also supported, including in multiple formats:

```
$ clipp \
raw:request1.http,response1.http \
raw:request2.http,response2.http \
ironbee:ironbee.conf
```

Modifiers are also supported, such as changing the IP/port used:

```
$ clipp \
raw:request1.http,response1.http \
@set_local_ip:1.2.3.4 \
@set_local_port:8080 \
raw:request2.http,response2.http \
@set_local_ip:5.6.7.8 \
@set_local_port:80 \
ironbee:ironbee.conf
```

With many parameters, the command line can get tedious, so clipp also supports a configuration file format:

```
### clipp.conf

# Transaction 1
raw:request1.http,response1.http
  @set_local_ip:1.2.3.4
  @set_local_port:8080

# Transaction 2
raw:request2.http,response2.http
  @set_local_ip:5.6.7.8
  @set_local_port:80

# IronBee Consumer
ironbee:ironbee.conf
```

```
$ clipp -c clipp.conf
```

Clipp's native protobuf format can encapsulate all input, including modifiers, into a single compact format. This format is produced by using the writepb consumer:

```
$ clipp \
raw:request1.http,response1.http \
raw:request2.http,response2.http \
writepb:input.pb
```

Full documentation on clipp is in Markdown format in the source tree: https://github.com/ironbee/ironbee/blob/master/clipp/clipp.md

Developer Module

IronBee includes a developer module that contains some diagnostic features. One that can aide in diagnosing inspection issues is the `TxDump` directive. This directive allows for dumping some internal IronBee state to a log file. To use `TxDump`, you need to load the `devel` module.

```
LoadModule "ibmod_devel.so"
```

Once this is done, you can use the `TxDump` directive to log data to a file.

```
TxDump TxFinished file:///tmp/ironbee_diag.log All
```

And you will get something like the following.

```
[TX 6bcb584c-7f84-461a-93b4-d576e7fd72a7 @ tx_finished_event]
  Started = 2013-04-25T17:34:00.2508-0700
  Hostname = waf-test.example.com
```

```
  Effective IP = 5.6.7.8
  Path = /
  Context
    Name = any:location:/
    Site name = any
    Site ID = 0CA1665C-F27F-4763-A3E0-A31A00477497
    Location path = /
  Connection
    Created = 2013-04-25T17:34:00.2508-0700
    Remote = 5.6.7.8:80
    Local = 1.2.3.4:1234
    Context
      Name = main:site:any
      Site name = any
      Site ID = 0CA1665C-F27F-4763-A3E0-A31A00477497
  Request line:
    Raw = "GET /path/test?foo=bar HTTP/1.1"
    Method = "GET"
    URI = "/path/test?foo=bar"
    Protocol = "HTTP/1.1"
  Request Header
    Host = "waf-test.example.com"
    User-Agent = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:18.0) Gecko/..."
    Accept = "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
    Accept-Language = "en-US,en;q=0.5"
    Connection = "keep-alive"
  Response line:
    Raw = "HTTP/1.1 301 Moved Permanently"
    Protocol = "HTTP/1.1"
    Status = "301"
    Message = "Moved Permanently"
  Response Header
    Location = "http://www.example.com/"
    Content-Type = "text/html; charset=UTF-8"
    Date = "Wed, 26 Oct 2012 02:36:41 GMT"
    Expires = "Fri, 25 Nov 2012 02:36:41 GMT"
    Cache-Control = "public, max-age=2592000"
    Server = "Apache"
    Content-Length = "219"
    X-XSS-Protection = "1; mode=block"
    X-Frame-Options = "SAMEORIGIN"
  Flags = 0380beb8
    00000001 "Error" = Off
    00000002 "HTTP/0.9" = Off
    00000004 "Pipelined" = Off
    00000008 "Request Started" = On
    00000020 "Seen Request Header" = On
    00000040 "No Request Body" = Off
    00000080 "Seen Request Body" = On
    00000100 "Seen Request Trailer" = Off
    00000200 "Request Finished" = On
```

```
    00000400 "Response Started" = On
    00001000 "Seen Response Header" = On
    00002000 "Seen Response Body" = On
    00004000 "Seen Response Trailer" = Off
    00008000 "Response Finished" = On
    00010000 "Suspicious" = Off
    00020000 "Block: Advisory" = Off
    00040000 "Block: Phase" = Off
    00080000 "Block: Immediate" = Off
    00100000 "Allow: Phase" = Off
    00200000 "Allow: Request" = Off
    00400000 "Allow: All" = Off
    00800000 "Post-Process" = On
    02000000 "Inspect Request Header" = On
    04000000 "Inspect Request Body" = Off
    08000000 "Inspect Response Header" = Off
    10000000 "Inspect Response Body" = Off
  ARGS:
    ARGS = [1]
    ARGS:foo = "bar"
  Data:
    response_line = "HTTP/1.1 301 Moved Permanently"
    request_uri = "/path/test?foo=bar"
    request_protocol = "HTTP/1.1"
    request_uri_params = [1]
      request_uri_params:foo = "bar"
    FIELD_NAME = ""
    request_content_type = ""
    request_uri_path = "/path/test"
    remote_addr = "5.6.7.8"
    request_host = ""
    request_filename = ""
    response_headers = [9]
      response_headers:Location = "http://www.example.com/"
      response_headers:Content-Type = "text/html; charset=UTF-8"
      response_headers:Date = "Wed, 26 Oct 2012 02:36:41 GMT"
      response_headers:Expires = "Fri, 25 Nov 2012 02:36:41 GMT"
      response_headers:Cache-Control = "public, max-age=2592000"
      response_headers:Server = "Apache"
      response_headers:Content-Length = "219"
      response_headers:X-XSS-Protection = "1; mode=block"
      response_headers:X-Frame-Options = "SAMEORIGIN"
    request_uri_password = ""
    request_uri_scheme = ""
    request_uri_query = "foo=bar"
    remote_port = 80
    request_line = "GET /path/test?foo=bar HTTP/1.1"
    server_addr = "1.2.3.4"
    response_status = "301"
    response_message = "Moved Permanently"
    CAPTURE = [0]
```

```
    FIELD_NAME_FULL = ""
    request_uri_username = ""
    request_body_params = [0]
    server_port = 1234
    response_cookies = [0]
    request_uri_port = ""
    response_protocol = "HTTP/1.1"
    conn_tx_count = 1
    request_uri_raw = "/path/test?foo=bar"
    CA = [0]
    auth_password = ""
    request_headers = [5]
      request_headers:Host = "waf-test.example.com"
      request_headers:User-Agent = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:18.0) Gecko/2010
      request_headers:Accept = "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
      request_headers:Accept-Language = "en-US,en;q=0.5"
      request_headers:Connection = "keep-alive"
    request_method = "GET"
    response_content_type = ""
    request_uri_fragment = ""
    request_uri_path_raw = "/path/test"
    ARGS = [1]
      ARGS:foo = "bar"
    FLAGS = [5]
      FLAGS:suspicious = 0
      FLAGS:inspectRequestHeader = 1
      FLAGS:inspectRequestBody = 0
      FLAGS:inspectResponseHeader = 0
      FLAGS:inspectResponseBody = 0
    auth_username = ""
    auth_type = ""
    request_uri_host = ""
    request_cookies = [0]
```

What data you log can be tailored. See the documentation for the `TxDump` directive for the full syntax.

Chapter 5: Inspection

The whole point of IronBee is to allow for inspecting data. This chapter covers the basics of doing just that. Inspecting data is performed through rules and modules. Rules are encapsulated logic executed at well define points in the transaction lifecycle. Modules, in general, extend IronBee functionality and can provide extensive logic, however modules can also be used for inspection when you need more control and flexibility.

Data

IronBee exposes data in two forms. Data can be in defined as a field or as a stream. In addition to inspection, fields are the primary means of exposing and exchanging data between rules and modules. A full list of the defined data fields are discussed in Chapter 6, however IronBee allows for the creation and modification of arbitrary fields. New variable fields can be defined, initialized in various ways within the IronBee configuration as well as in rules and modules. This section discusses defining and initializing rules via the configuration, leaving further discussion for the Rules section.

Variable fields, or just variables for short, are defined and initialized in the the configuration through two primary directives. The scope of these variables is defined through the context. Variables can be defined in the main (global) scope, within a site scope or within a location scope based on where the directives are used in the configuration.

Defining and Initializing Scalar Variables with `InitVar`

A simple scalar variable is both defined and initialized with the `InitVar` directive.

```
InitVar MY_VAR "my value"
```

In addition to simple scalar variables, you can also use `InitVar` to define scalar variables within a collection. If the collection does not previously exist, then it is created.

```
InitVar MY_COLLECTION:MY_VAR "my value"
```

The variable will then be available as a normal field to all rules and modules active in the same configuration context (scope). See the full documentation for the `InitVar` directive for more details.

Defining and Initializing Scalar Variables with `InitVarIndexed`

As `InitVar`, but also indicates that the variable should be indexed. Indexed variables can be accessed more quickly but impose a small memory cost on every transaction.

Defining and Initializing Collections with `InitCollection`

Similar to the `InitVar` directive, you can also define and initialize collections with the `InitCollection` directive.

```
InitCollection MY_COLLECTION
```

The above will define the `MY_COLLECTION` collection, however, it will be empty. To both define and initialize a collection, you can utilize a number of additional `InitCollection` features. The most basic form allows specifying key/value pairs with the collection.

```
InitCollection MY_COLLECTION vars: \
    key1=value1 \
    key2=value2 \
    key3=value3
```

In addition to initializing the key values within the configuration file, you may also initialize the collection via an external JSON formatted file, allowing for controlling the data withing the collection outside of IronBee.

```
InitCollection MY_COLLECTION json-file:///path/to/persisted/data/mycollection.json
```

The JSON is formatted as simple name/value pairs.

```
{
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

`InitCollectionInexed` is also available. See `InitVarIndexed`.

See the full documentation for `InitCollection` directive for more details.

Persisting Data with `InitCollection`

In addition to defining and initializing collections with static data, IronBee also allows for persisting collections which have been manipulated in IronBee. This is accomplished in a few different ways, depending on your needs. If you need to persist a single instance of a collection, then you can just add the persist option to the InitCollection directive.

```
InitCollection MY_COLLECTION json-file:///path/to/persisted/data/mycollection.json persist
```

With the addition of the persist option, any data within the collection will be written out to the JSON file at the end of the transaction. The next transaction will then be initialized with the manipulated data. Often, though, you do not want to share the same collection for all

transactions. Instead you need to be able to save different instances of the collection based on some other field or set of fields as a key. To do this, you need to load the persist module to gain some additional functionality.

```
LoadModule ibmod_persist.so
```

The persist module allows you to store different instances of a collection based on a key. For example, you may want to store an instance of the collection based on the IP address or session ID. This is not any more difficult, just a slightly different syntax. The key can be specified using any string, which may include field expansion, such as the IP address or session ID.

```
InitCollection IP_DATA persist-fs:///path/to/persisted/data key=%{REMOTE_ADDR}
InitCollection SESS_DATA persist-fs:///path/to/persisted/data key=%{REQUEST_COOKIES:jsessionid}
```

Any data contained in these collections will be read from a file based on the key and stored in the named collection. At the end of the transaction, the data is written out to the same file. Since this data may accumulated, you will probably want to specify an expiration time. This is done by using the expire option, which takes an expiration time in seconds. If more than the number of seconds elapses between the collection being written out and read back in, the data is purged and the collection will be empty.

```
InitCollection IP_DATA persist-fs:///path/to/persisted/data key=%{REMOTE_ADDR} expire=300
```

Since the data is only purged when it is attempted to be read back in after expiring, the data may still accumulate on the filesystem. It may be required to run a periodic cleanup process to purge any expired files. In the future IronBee will provide a utility for this, but for now the expiration date is encoded in the filename.

```
# Format: uuid/expiration-tempname
0de114da-8ada-55ad-a6de-e68a1263412a/001364624257-0004d91e578bc99f.json.dXFR9d
```

Periodic purging could be accomplished with a cron job to check that the current epoch based date is greater than that encoded in the file.

```
#!/bin/sh

# Specify the persist-fs: base directory
PERSIST_FS_BASEDIR="/tmp/ironbee/persist/fs"

# Current epoch based date
DSTAMP=`date "+%s"`

# Iterate through files
for file in `find $PERSIST_FS_BASEDIR -type f -name '*.json.*'`; do
    # Extract the epoch based expiration from the filename
    expires=`echo $file | sed 's%.*/0*\([0-9]*\)-.*%\1%'`
```

```
    # Check if the expires was extracted and the current date
    # is greater than the expiration, removing the file.
    if [ -n "$expires" -a "$DSTAMP" -gt "$expires" ]; then
        echo "PURGE: $file expired=`date -j -r $expires`"
        rm $file
    fi
done
```

Rules

Rules are the primary form of inspection in IronBee. IronBee rule execution is decoupled from any rule language. Because of this, IronBee can provide multiple rule languages. Each language has a different use case. Currently the following rule languages are defined:

- IronBee Rule Language, which is part of the IronBee Configuration Language.
- Lua rule definitions, available in Lua modules and Lua configuration files.
- External Lua rule scripts.
- Alternative rule execution via rule injection modules.

IronBee Rule Language

The IronBee rule language is relatively simplistic. The language is designed to create signature based rules with minimal logic. If you need more logic, then you should consider other options.

The rule language allows for inspecting fields and performing actions. There are three forms of rules:

- Field based inspection rules which execute actions based on inspecting a set of fields.
- Stream based inspection rules which execute actions based on inspecting a stream of data.
- Actions based rules, which just execute actions and allow for some basic logic and setup.

Inspecting Fields with the `Rule`

The Rule directive allows inspecting a set of fields and optionally executing an action. For example, you can specify a list of request methods that you wish to block.

```
Rule REQUEST_METHOD @imatch "TRACE TRACK" \
    id:test/methods/1 \
    phase:REQUEST_HEADER \
    "msg:Invalid method: %{REQUEST_METHOD}" \
    event:alert \
    block:phase
```

The example above inspects the `REQUEST_METHOD` field using the `@imatch` operator. The `@imatch` operator matches case insensitively against a list of values. In this case the match is a success if the `REQUEST_METHOD` completely matches any of the specified methods. If the match is a success,

then the event and block actions will be executed, logging an alert with the given message and blocking the request at the end of the phase. There are a few additional modifiers. The id and phase modifiers are required. The id modifier must be a unique string and the phase modifier specifies when the rule will execute. In this case the rule will execute just after the HTTP request headers are available.

As an alternate to the above, you could instead whitelist what methods you wish to allow with a similar rule. In this case you would just negate the operator and specify a list of methods that are allowed. If the method is not on the list, then the actions will execute.

```
Rule REQUEST_METHOD !@imatch "GET HEAD POST" \
    id:test/methods/1 \
    phase:REQUEST_HEADER \
    "msg:Invalid method: %{REQUEST_METHOD}" \
    event:alert \
    block:phase
```

More than one field can be specified. If so, then each value will be run through the operator, triggering actions for each match. In addition, the field values can be transformed, such as trimming off any whitespace.

```
Rule REQUEST_METHOD.trim() !@imatch "GET HEAD POST" \
    id:test/methods/1 \
    phase:REQUEST_HEADER \
    "msg:Invalid method: %{REQUEST_METHOD}" \
    event:alert \
    block:phase
```

Transformations can be specified per-field, or to all fields, using, for example, the `t:trim` rule modifier. Multiple transformations can be chained together.

See the `Rule` directive documentation for more details.

Inspecting Streams with `StreamInspect`

Potentially large fields, such as the request and response body, pose problems when they need to be inspected as a whole. To alleviate problems with requiring large amounts of memory for inspection, the request and response bodies are only available as streams. The `StreamInspect` directive is used to write stream based data. This directive differs slightly from the Rule directive.

- `StreamInspect` rules run as data is received, which is before phase rules execute on the request/response bodies. Any setup with phase based rules should be done in the associated header phase to ensure they are executed before stream based rules. Depending on the size of the data and the server's buffer size, the data may come in chunks. Because of this, a `StreamInspect` rule may execute multiple times - once per chunk of data received.
- `StreamInspect` rules have a limited set of operators that support streaming inspection. Currently this is limited to the `dfa` operator, but may expand in the future. The `dfa` operator

uses the PCRE syntax similar to `rx`, but does not allow backtracking. Additionally, the `dfa` operator can capture ALL matches, instead of just the first as `rx` does. This allows capturing all matching patterns from the stream. Note that the `dfa` operator is fully streaming aware and will match across chunk boundaries.

- `StreamInspect` rules allow only a single stream as input, however you can use multiple rules.
- `StreamInspect` rules currently do not support transformations.

See the `StreamInspect` documentation for more details.

Executing actions with `Action`

Rule actions may need to be triggered unconditionally. While not often required, this is possible with the `Action` directive. Typically this is used to execute `setvar`, `setflag` or similar actions.

```
Action id:init/1 phase:REQUEST_HEADER setvar:MY_VAR=1234
```

See the `Action` documentation for more details.

Lua Signature Definitions

Often you may need more functionality in configuring rules than is offered by the configuration language. This is possible by using Lua to provide signature definitions. Using the `LuaInclude` directive, you can include a lua script into the configuration. The Lua script can define rules as an alternate signature definition language. Note that Lua is only being used as the configuration language. This means that Lua is only executed at configuration time and not required to execute the rules. The rules defined in the lua script are identical to those added via the Rule directive, but just use an alternative configuration language. This really shows off IronBee's separation of the rules from the language in which they are defined.

```
# Load the Lua module to add Lua functionality into IronBee.
LoadModule ibmod_lua.so

# Include rules via a lua script and commit.
LuaInclude rules.lua
LuaCommitRules
```

Including a lua script at configuration using `LuaInclude` allows the full power of Lua to configure the rules. The included Lua script is executed at config time, providing a vast amount of power over rule configuration. Within Lua, you can use the `Sig(id,rev)` function to define signature rules. The `Sig()` function returns a signature object, which allows you to then specify attributes, such as fields, an operator, actions, etc. The following is a simple rule using the `Rule` directive, which will serve as an example to be converted using the Lua configuration.

```
Rule ARGS REQUEST_HEADERS \
```

```
     @rx "some-attack-regex" \
     id:test/lua/1 rev:1 \
     severity:50 confidence:75 \
     event:alert block:phase \
     "msg:Some message text."
```

This is converted into Lua's `Sig()` function below. Note that this is an extremely verbose version for clarity. Later, this will be shortened to a much more manageable form.

```
-- Create a signature with: id="test/lua/1" rev=1
local sig = Sig("test/lua/1", 1)

-- Specify what fields to inspect.
sig:fields("ARGS", "REQUEST_HEADERS")

-- Specify the phase.
sig:phase("REQUEST")

-- Specify the operator
sig:op("rx", [[some-attack-regex]])

-- Specify other meta-data.
sig:action("severity:50")
sig:action("confidence:75")

-- Specify the actions.
sig:action("event:alert")
sig:action("block:phase")
sig:message("Some message text.")
```

The `Sig()` function returns a signature object as do all the attribute functions. This allows us to chain attributes via the colon operator resulting in something much more compact and "rule-like".

```
Sig("test/lua/1", 1):
  fields("ARGS", "REQUEST_HEADERS"):
  phase("REQUEST"):
  op("rx", [[some-attack-regex]]):
  action("severity:50"):
  action("confidence:75"):
  action("event:alert"):
  action("block:phase"):
  message("Some message text.")
```

Even this, however, is a bit more verbose than desired. In practice many rules will follow the same form and it will quickly become tedious to write signatures in such a verbose format. To reduce this verbosity, the power of Lua is utilized, which allows customizing how rules are written by defining wrapper functions around the default `Sig()` function.

```
--[[ ------------------------------------------------
```

```
---- Define a function to reduce verbosity:
---- RequestRegex(id, regex [,severity [,confidence]])
--]] ----------------------------------------------------
local RequestRegex = function(id,regex,severity,confidence)
  if severity == nil then
    severity = 50
  end
  if confidence == nil then
    confidence = 75
  end
  return Sig("test/lua/" .. id,1):
           op("rx", regex):
           phase("REQUEST"):
           action("severity:" .. severity):
           action("confidence:" .. confidence):
           action("event:alert"):
           action("block:phase")
end

--[[ ----------------------------------------------------
---- Define a list of common attack fields
--]] ----------------------------------------------------
local ATTACK_FIELDS = { "ARGS", "REQUEST_HEADERS" }

-- Rules using the above wrappers
RequestRegex(1,[[some-attack-regex]]):
  fields(ATTACK_FIELDS):
  message("Some message text.")
```

As you can see, this can substantially reduce the verbosity of the rules, however, it does require writing some wrapper functions. As IronBee matures, it will expose some builtin wrappers in a separate library. Separating the wrappers into a library would then reduce this into a file that load the library alongside the rules themselves.

```
-- Load the Wrappers
require rule-wrappers

-- Rules
RequestRegex(1,[[some-attack-regex]]):
  fields(ATTACK_FIELDS):
  message("Some message text.")
RequestRegex(2,[[some-other-attack-regex]]):
  fields(ATTACK_FIELDS):
  message("Some other message text.")
```

Rule execution order is different when specified in Lua. In Lua, no order is guaranteed unless specified. Order is specified in a number of ways. The first method is via the `before()` or `after()` attributes, which control rule execution order. Note that `before()` and `after()` are not rule chaining and do not require the previous rule to match.

```
Sig("lua/1",1):
  before("lua/2")
Sig("lua/2",1):
Sig("lua/3",1):
  after("lua/2")
```

While this is powerful, it is tedious to maintain. As most cases where you need rule order are in grouping rules to form a sort of recipe, there is a `Recipe(tag)` function defined which does the following:

- Adds the supplied recipe tag to all rules within the recipe.
- Forces rule execution order within the recipe.

```
Recipe "recipe/1" {
  Sig("lua/1",1),
  Sig("lua/2",1),
  Sig("lua/3",1)
}
```

Each rule in the recipe will contain the recipe tag and therefore the entire recipe can be enabled via the `RuleEnable` directive.

```
RuleEnable tag:recipe/1
```

The `Rule` directive supports chaining rules via the `chain` rule modifier. Chaining allows rules to be logically ANDed together so that later rules only execute if previous rules match. Chained rules are slightly different when specified in Lua. Lua uses the `follows()` attribute to specify a rule ID to follow in execution IF that rule matches. This is essentially reversed from the `Rule` directive which specifies the `chain` modifier on the previous rule verses specifying the `follows()` attribute on the later rule.

```
# Define a "lua/1" rule
Sig("lua/1",1)

# Define a "lua/2" rule that will run only if "lua/1" matches
Sig("lua/2",1):follows("lua/1")

# Define a "lua/3" rule that will run only if "lua/2" matches
Sig("lua/3",1):follows("lua/2")
```

The following is defined for use in defining rules within Lua.

- Sig(id,rev): Create a new signature based rule.
  - field(name): Specify a single field name added to the list of fields to inspect.
  - fields(list): Specify a list of field names to be added to the list of fields to inspect.
  - op(name,value): Specify an operator to use for the rule.

    - phase(name): Specify the phase name to execute within.
    - message(text): Specify a message for the rule.
    - tag(name): Specify a tag name to add to the list of tags.
    - tags(list): Specify a list of tag names to be added to the list of tags.
    - comment(text): Arbitrary comment text to associate with the rule.
    - action(text): Specify any additional rule action or modifier in "name:parameter" format.
    - before(rule-id): Specify the rule ID which this should execute before.
    - after(rule-id): Specify the rule ID which this should execute after.
    - follows(rule-id): Specify the rule ID that this should follow IF that rule matched.
- Action(id,rev): Similar to the Action directive, this is the same as Sig(), but disallows field()/fields()/op() attributes.
- ExtSig(id,rev): Similar to the RuleExt directive, this is the same as Sig(), but allows specifying a script to execute as the rule logic.
    - script(name): Name of script to execute.
- Recipe(tag, rule-list): Group a list of rules, adding tag to all rules and maintaining rule execution order.

External Lua Rule Scripts

While Lua signature definitions are very powerful, they are still limited to signature like operations. To allow for complex logic you can use Lua at rule execution time yielding the full power of Lua as an inspection language. This is accomplished by using either the `RuleExt` directive within a configuration file or `ExtSig()` within a Lua configuration file.

See the documentation for the `RuleExt` directive for more details.

Alternative Rule Execution via Rule Injection Modules

Modules may define additional rule execution systems via the rule injection mechanism. Rule injection works in two stages:

- At the end of configuration, every rule injection system is given a chance to claim each rule. Rule injection systems usually claim a rule if it contains a certain action. Only one rule injection system may claim each rule; it is an error for more than one to claim it. If no rule injection system claims a rule, it is added to the default rule engine.
- At each phase during inspection, every rule injection system is given a chance to inject one or more rules. The rule injection system may use whatever method it desires to choose which rules to inject. Injected rules are then executed as usual.

The rule injection mechanism is designed to allow for specialized rule systems that, for a certain class of rules, are more expressive, more performant or both. For example, the Fast rule injection

systems associates a substring pattern with a rule and uses an Aho-Corasick variant to determine which rules to inject. The benefit over the traditional rule system is that rules that do not fire have minimal performance cost. However, Fast is only suitable for a subset of rules: those that require certain fixed width patterns to appear in the input.

The default rule engine claims all rules not otherwise claimed. It evaluates each rule for the appropriate phase and context in order. This approach is slow but also simple and predictable.

Modules

When full control is required, then an IronBee module may be required. Modules provide the ability to hook directly into the IronBee state machine for fine grained control over execution, Currently modules can be written in three languages. Each has a different use case which is described below.

- Lua is the simplest language to develop modules as it hides many of the details. While Lua allows for rapid development, it does not perform as well as other languages for many tasks. Lua is the recommended language for prototyping and most higher level module needs - where Lua rules are not adequate. Lua modules also have the added benefit of being able to be distributed as rules, since they are not in a binary form.
- C++ allows near full control over IronBee via the C++ wrappers. C++ provides much higher level access to IronBee in a fairly strict environment. However, the C++ wrappers do not cover all functionality of IronBee and you may need to fall back to the C API. Because of the added strictness in C++ and near equal performance to the native C API, it is the recommended language if Lua will not satisfy performance or functionality requirements.
- C is the lowest level language for writing modules. While C provides full functionality, it does not provide as much protection as C++ or Lua.

See Chapter 7 for more information on writing IronBee modules.

Chapter 6: Rule Reference

...

Data Fields

...

ARGS

Description: All request parameters combined and normalized.

Type: Collection

Scope: Transaction (`REQUEST_HEADERS`, `REQUEST_BODY`)

Module: core

Version: 0.2

> **Note**
>
> The `ARGS` collection is currently the same as specifying `REQUEST_URL_PARAMS REQUEST_BODY_PARAMS`, but this will change in later releases to include normalization based on parser personalities. If you do not want normalization, then use `REQUEST_URL_PARAMS REQUEST_BODY_PARAMS`.

AUTH_PASSWORD

Description: Basic authentication password.

Type: String

Scope: Transaction

Module: core

Version: 0.7

AUTH_TYPE

Description: Indicator of the authentication method used.

Type: Collection

Scope: Transaction

Module: core

Version: 0.7

This field contains the first token extracted from the `Authorization` request header. Typical values are: `Basic`, `Digest`, and `NTLM`.

AUTH_USERNAME

Description: Basic or Digest authentication username.

Type: String

Scope: Transaction

Module: core

Version: 0.7

CAPTURE

Description: Transaction collection.

Type: Collection

Scope: Transaction

Module: core

Version: 0.4

This collection contains information for the transaction. Currently captured data from operators is stored here in keys "0"-"9".

FIELD

Description: An alias to the current field being inspected.

Type: Variable (same type as the aliased field)

Scope: Rule

Module: core

Version: 0.5

This field is useful only in field expansions within actions when you must have the original value of the field being inspected. For example:

```
# Log the field value with an event
Rule ARGS @contains attack_string id:123 phase:REQUEST logdata:%{FIELD} event

# Create a collection matching a pattern for later use
Rule REQUEST_HEADERS @rx pattern1 id:124 phase:REQUEST_HEADER setvar:NEW_COL:%{FIELD_NAME}=%{FIELD}
Rule ARGS @rx pattern2 id:125 phase:REQUEST setvar:NEW_COL:%{FIELD_NAME}=%{FIELD}
...
# Then perform further matches on the new collection in another phase, which
# is not possible via chaining.
Rule NEW_COL @rx some_other_patt id:126 phase:REQUEST "msg:Some msg" event block
```

FIELD_NAME

Description: An alias to the current field name being inspected, not including the collection name if it is a sub-field in a collection.

Type: Variable (same type as the aliased field)

Scope: Rule

Module: core

Version: 0.5

This field is useful only in field expansions within actions when you must have the name of the field being inspected. The collection name is not prepended, so if `ARGS:foo` is being inspected, the value will be `foo`, not `ARGS:foo`. If you want the full name with the collection prepended, then use `FIELD_NAME_FULL`.

FIELD_NAME_FULL

Description: An alias to the current field name being inspected, including the collection name if it is a sub-field in a collection.

Type: Variable (same type as the aliased field)

Scope: Rule

Module: core

Version: 0.5

This field is useful only in field expansions within actions when you must have the full name of the field being inspected. See `FIELD_NAME`.

GEOIP

Description: If the *geoip* module is loaded, then a lookup will be performed on the remote (client) address and the results placed in this collection.

Type: Collection

Scope: Transaction

Module: geoip

Version: 0.3

> **Note**
>
> The address used during lookup is the same as that stored in the `REMOTE_ADDR` field, which may be modified from the actual connection (TCP) level address by the `user_agent` module.

Sub-Fields (not all are available prior to GeoIP v1.4.6):

- **latitude:** Numeric latitude rounded to nearest integral value (no floats yet).
- **longitude:** Numeric longitude rounded to nearest integral value (no floats yet).
- **area_code:** Numeric area code (US only).
- **charset:** Numeric character set code.

- **country_code:** Two character country code.
- **country_code3:** Three character country code.
- **country_name:** String country name.
- **region:** String region name.
- **city:** String city name.
- **postal_code:** String postal code.
- **continent_code:** String continent code.
- **accuracy_radius:** Numeric accuracy radius (v1.4.6+).
- **metro_code:** Numeric metro code (v1.4.6+).
- **country_conf:** String country confidence (v1.4.6+).
- **region_conf:** String region confidence (v1.4.6+).
- **city_conf:** String city confidence (v1.4.6+).
- **postal_conf:** String postal code confidence (v1.4.6+).

HTP_REQUEST_FLAGS

Description: Collection of LibHTP request parsing flags.

Type: Collection

Scope: Transaction

Module: htp

Version: 0.3

The LibHTP parser will set various flags while parsing. This is a collection of those flags for request parsing. The following flags may be set:

- **FIELD_UNPARSEABLE** An unparseable field was given.
- **FIELD_INVALID** An invalid field was sent.
- **FIELD_FOLDED** Folding detected in a field.
- **FIELD_REPEATED** A field was repeated.
- **FIELD_LONG** A field length was longer than allowed.
- **FIELD_RAW_NUL** A field contained an unencoded NUL (zero) byte.
- **HOST_AMBIGUOUS** The host was specified in both the URI and in the Host header, but they do not match.
- **HOST_MISSING** The host was missing from a request in which it is normally sent.
- **HOSTH_INVALID** Invalid host detected in header.
- **HOSTU_INVALID** Invalid host detected in URL.
- **INVALID_FOLDING** Invalid header folding detected.

- **INVALID_CHUNKING** Invalid chunking detected.
- **MULTI_PACKET_HEAD** The header was sent in more than one packet (buffer).
- **PATH_ENCODED_NUL** A NUL (zero) byte was sent, encoded, in the path.
- **PATH_ENCODED_SEPARATOR** An encoded path separator was sent in the path.
- **PATH_HALF_FULL_RANGE** An invalid full width character was used in the path.
- **PATH_INVALID** An invalid path detected.
- **PATH_INVALID_ENCODING** Invalid encoding was used in the path.
- **PATH_OVERLONG_U** An overlong Unicode encoding was used in the path.
- **PATH_UTF8_VALID** A UTF-8 character was used in the path.
- **PATH_UTF8_INVALID** An invalid UTF-8 encoding was used in the path.
- **PATH_UTF8_OVERLONG** An overlong UTF-8 encoding was used in the path.
- **REQUEST_SMUGGLING** A HTTP smuggling attack was detected.
- **URLEN_ENCODED_NUL** An encoded NUL (zero) byte detected in URL.
- **URLEN_HALF_FULL_RANGE** An invalid full width character detected in URL.
- **URLEN_INVALID_ENCODING** An invalid encoding detected in URL.
- **URLEN_OVERLONG_U** An overlong unicode character detected in URL.

HTP_RESPONSE_FLAGS

Description: Collection of LibHTP response parsing flags.

Type: Collection

Scope: Transaction

Module: htp

Version: 0.3

The LibHTP parser will set various flags while parsing. This is a collection of those flags for response parsing. The following flags may be set:

- **FIELD_UNPARSEABLE** An unparseable field was given.
- **FIELD_INVALID** An invalid field was sent.
- **FIELD_FOLDED** Folding detected in a field.
- **FIELD_REPEATED** A field was repeated.
- **FIELD_LONG** A field length was longer than allowed.
- **FIELD_RAW_NUL** A field contained an unencoded NUL (zero) byte.
- **INVALID_CHUNKING:** Invalid chunking was used.
- **INVALID_FOLDING:** Invalid header folding was used.

- **MULTI_PACKET_HEAD:** The header was sent in more than one packet (buffer).
- **STATUS_LINE_INVALID:** An invalid HTTP status code was sent.

REMOTE_ADDR

Description: Remote (client) IP address, extracted from the TCP connection. Can be in IPv4 or IPv6 format.

Type: String

Scope: Connection

Module: core

Version: 0.2

> **Note**
>
> If the `user_agent` module is also loaded, then the client address will be corrected using any available proxy headers (currently `X-Forwarded-For`).

REMOTE_PORT

Description: Remote (client) port, extracted from the TCP connection.

Type: Numeric

Scope: Connection

Module: core

Version: 0.2

REQUEST_BODY_PARAMS

Description: Request parameters transported in request body.

Type: String

Scope: Transaction

Module: core

Version: 0.4

REQUEST_CONTENT_TYPE

Description: Contains the normalized request content type.

Type: String

Scope: Transaction (`REQUEST_HEADERS`)

Module: core

Version: Not implemented yet

Request content type is constructed from the request `Content-Type` header. The value is first converted to contain only the content type (and exclude any character encoding information), then converted to lowercase.

REQUEST_COOKIES

Description: Collection of request cookies (name/value pairs).

Type: Collection

Scope: Transaction (`REQUEST_HEADERS`)

Module: core

Version: 0.2

REQUEST_FILENAME

Description: Request filename, extracted from request URI and normalized according to the current personality.

Type: String

Scope: Transaction

Module: core

Version: Not implemented yet

Normalization algorithm, with all "features" enabled, is as follows:

1. Decode URL-encoded characters (both `%HH` and `%uHHHH` formats), convert to lowercase, compress separators, convert backslashes, and terminate NUL.
2. Convert UTF-8 to single-byte stream using best-fit mapping
3. Perform RFC 3986 normalization

REQUEST_HEADERS

Description: Collection of request headers (name/value pairs).

Type: Collection

Scope: Transaction (`REQUEST_HEADERS`)

Module: core

Version: 0.2

REQUEST_HOST

Description: Request hostname information, extracted from the request and normalized.

Type: String

Scope: Transaction (`REQUEST_HEADERS`)

Module: core

Version: 0.2

The following rules apply:

1. Use the hostname information if provided on the request line
2. Alternatively, look up the HTTP `Host` request header
3. If the hostname information is provided in both locations, the information in the HTTP `Host` request header is ignored

Normalization [TODO What RFC should we refer to?]:

1. Lowercase
2. Remove trailing dot [TODO What dot?]
3. [TODO Remove port?]

REQUEST_LINE

Description: Full, raw, request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

Example:

```
GET /path/to/page?a=5&q=This+is+a+test. HTTP/1.1
```

REQUEST_METHOD

Description: Request method.

Type: String

Scope: Transaction

Module: core

Version: 0.3

This field contains the HTTP method used for the request.

Example: `GET`

REQUEST_PROTOCOL

Description: Request protocol name and version.

Type: String

Scope: Transaction

Module: core

Version: o.3

This field contains the HTTP protocol name and version, as specified on the request line. Transactions that do not specify the protocol (e.g., HTTP prior to 1.0) will have an empty string value.

REQUEST_URI

Description: Request URI, extracted from request and normalized according to the current personality (see `REQUEST_FILENAME` for more details).

Type: String

Scope: Transaction

Module: core

Version: 0.2

Default normalization:

1. RFC normalization
2. Convert to lowercase
3. Reduce consecutive forward slashes to a single character

All normalization options:

- RFC normalization
- Convert to lowercase
- Convert \ characters to /
- Reduce consecutive forward slashes to a single character

REQUEST_URI_FRAGMENT

Description: Parsed fragment portion of the URI within the request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

REQUEST_URI_HOST

Description: Parsed host portion of the URI within the request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

This is the hostname specified in the URI. Note that this may be different from the normalized host, which is in `REQUEST_HOST`.

REQUEST_URI_PARAMS

Description: Request parameters transported in query string.

Type: Collection

Scope: Transaction (`REQUEST_HEADERS`)

Module: core

Version: 0.2

REQUEST_URI_PASSWORD

Description: Parsed password portion of the URI within the request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

REQUEST_URI_PATH

Description: Parsed and normalized path portion of the URI within the request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

REQUEST_URI_PATH_RAW

Description: Parsed (raw) path portion of the URI within the request line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

> **Note**
>
> As no URL decoding is performed (this is a raw value), you probably want `REQUEST_URI_PATH_RAW.urlDecode()` in most cases.

REQUEST_URI_PORT

Description: Parsed port portion of the URI within the request line.
Type: String
Scope: Transaction
Module: core
Version: 0.3

REQUEST_URI_RAW

Description: Raw, unnormalized, request URI from the request line.
Type: String
Scope: Transaction
Module: core
Version: 0.2

REQUEST_URI_SCHEME

Description: Parsed scheme portion of the URI within the request line.
Type: String
Scope: Transaction
Module: core
Version: 0.3

REQUEST_URI_QUERY

Description: Parsed query portion of the URI within the request line.
Type: String
Scope: Transaction
Module: core
Version: 0.3

REQUEST_URI_USERNAME

Description: Parsed username portion of the URI within the request line.
Type: String
Scope: Transaction
Module: core
Version: 0.3

RESPONSE_CONTENT_TYPE

Description: Contains the normalized response content type.

Type: Scalar

Scope: Transaction (`RESPONSE_HEADERS`)

Module: core

Version: Not implemented yet

Response content type is constructed from the response `Content-Type` header. The value is first converted to keep only the content type part (and exclude character encoding information, if any), then converted to lowercase.

RESPONSE_COOKIES

Description: Collection of response cookies (name/value pairs).

Type: Collection

Scope: Transaction

Module: core

Version: Not implemented yet

RESPONSE_HEADERS

Description: Collection of response headers (name/value pairs).

Type: Collection

Scope: Transaction

Module: core

Version: 0.2

RESPONSE_LINE

Description: Full response line.

Type: String

Scope: Transaction

Module: core

Version: 0.3

Transactions that do not specify a response line (e.g., HTTP prior to 1.0) will have an empty string value.

Example:

```
HTTP/1.1 200 OK
```

RESPONSE_MESSAGE

Description: Response status message.

Type: String

Scope: Transaction

Module: core

Version: 0.3

This field contains the status message (text following the status code), as specified on the response line. Transactions that do not specify a response line (e.g., HTTP prior to 1.0) will have an empty string value.

RESPONSE_PROTOCOL

Description: Response protocol name and version.

Type: String

Scope: Transaction

Module: core

Version: 0.3

This field contains the protocol name and version, as specified on the response line. Transactions that do not specify a response line (e.g., HTTP prior to 1.0) will have an empty string value.

RESPONSE_STATUS

Description: Response status code.

Type: String

Scope: Transaction

Module: core

Version: 0.3

This field contains the status code, as specified on the response line. Transactions that do not specify a response line (e.g., HTTP prior to 1.0) will have an empty string value.

SERVER_ADDR

Description: Server IP address, extracted from the TCP connection. Can be in IPv4 or IPv6 format.

Type: String

Scope: Connection

Module: core

Version: 0.2

SERVER_PORT

Description: Server port, extracted from the TCP connection.

Type: Numeric

Scope: Connection

Module: core

Version: 0.2

TX

Description: Transaction collection.

Type: Collection

Scope: Transaction

Module: core

Version: 0.3

This collection contains arbitrary information for the transaction. It is a generic place for rules to store transaction data in which other rules can monitor.

UA

Description: User agent information extracted if the `user_agent` module is loaded.

Type: Collection

Scope: Transaction

Module: user_agent

Version: 0.3

> **Note**
>
> While the `User-Agent` HTTP request header may be used in generating these fields, the term "user agent" here refers to the client as a whole.

Sub-Fields:

- **agent:** String name of the user agent.
- **product:** String product deduced from the user agent data.
- **os:** String operating system deduced from user agent data.
- **extra:** Any extra string available after parsing the `User-Agent` HTTP request header.
- **category:** String category deduced from user agent data.

Operators

...

contains

Description: Returns true if the target contains the given sub-string.

Syntax:`@contains "sub-string"`

Types: String

Module: core

Version: 0.3

dfa

Description: Deterministic finite atomation matching algorithm (PCRE's alternative matching algorithm).

Syntax:`@dfa "pcre-dfa-regex"`

Types: String

Module: pcre

Version: 0.4

The `dfa` operator implements the alternative matching algorithm in the PCRE [http://www.pcre.org/] regular expressions library. The parameter of the operator is a regular expression pattern that is passed to the PCRE library without modification. This alternative matching algorithm uses a similar syntax to PCRE regular expressions, except that backtracking is not available. The primary use of `dfa` is to allow a subset of regular expression matching in a streaming manner (see `StreamInspect`). In addition to streaming support, dfa will also find all matches to the pattern when the capture modifier is used. TODO: Describe limits on regex syntax.

Example of capturing multiple matches:

```
# Capture each item in a '&' separated list
Rule REQUEST_URI_QUERY @dfa "[^&]*" id:1 rev:1 phase:REQUEST_HEADER capture
# Inspect each element in the CAPTURE, blocking if the format does not match
Rule CAPTURE !@rx ".=." id:2 rev:1 phase:REQUEST_HEADER "msg:Name and value required" event block
```

ee_match_any

Description: Returns true if the target matches any value in the named eudoxus automata.

Syntax:`@ee_match_any automata-name`

Types: String

Module: ee

Version: 0.7

The named eudoxus automata must first be loaded with the `LoadEudoxus` directive

eq

Description: Returns true if the target is numerically equal to the given value.

Syntax:`@eq` *`number`*

Types: Numeric

Module: core

Version: 0.3

ge

Description: Returns true if the target is numerically greater than or equal to the given value.

Syntax:`@ge` *`number`*

Types: Numeric

Module: core

Version: 0.3

gt

Description: Returns true if the target is numerically greater than the given value.

Syntax:`@gt` *`number`*

Types: Numeric

Module: core

Version: 0.3

imatch

Description: As `match`, but case insensitive.

Syntax:`@imatch` *`"word1 word2 ... wordN"`*

Types: String

Module: core

Version: 0.7

ipmatch

Description: Returns true if a target IPv4 address matches any given whitespace separated address in CIDR format.

Syntax:`@ipmatch` *`"1.2.3.4 192.168.0.0/24 10.0.0.0/8"`*

Types: String

Module: core

Version: 0.3

ipmatch6

Description: Returns true if a target IPv6 address matches any given whitespace separated address in CIDR format.

Syntax:`@ipmatch6 "1::12:13 6::6:0/112 1::2:3"`

Types: String

Module: core

Version: 0.3

is_sqli

Description: Returns true if the data is determined to be SQL injection via the libinjection library.

Syntax:`@is_sqli data-source`

Types: String

Module: libinjection

Version: 0.7

The libinjection ironbee module utilizes Nick Galbreath's libinjection to implement SQLi detection. This operator is similar to libinjection's is_sqli() function. The libinjection library is available via: http://www.client9.com/projects/libinjection/

Currently the *`data-source`* must be set to "default" as loading external databases is not yet implemented.

Example:

```
Rule ARGS @is_sqli default id:test/sqli/1 phase:REQUEST "msg:Detected SQLi" logdata:%{FIELD} event b
```

le

Description: Returns true if the target is numerically less than or equal to the given value.

Syntax:`@le number`

Types: Numeric

Module: core

Version: 0.3

lt

Description: Returns true if the target is numerically less than the given value.

Syntax:`@lt number`

Types: Numeric

Module: core

Version: 0.3

match

Description: Returns true if the target is any of the given whitespace separated words.

Syntax:`@match "word1 word2 ... wordN"`

Types: String

Module: core

Version: 0.7

ne

Description: Returns true if the target is not numerically equal to the given value.

Syntax:`@ne number`

Types: Numeric

Module: core

Version: 0.3

pm

Description: Parallel matching using the `Aho-Corasick` algorithm.

Syntax:`@pm "word1 word2 ... wordN"`

Types: String

Module: ac

Version: 0.2

Implements a set-based (or parallel) matching function using the Aho-Corasick algorithm. The parameter of the operator contains one or more matching patterns, separated with whitespace. Set-based matching is capable of matching many patterns at the same time, making it efficient for cases when the number of patterns is very large (in hundreds and thousands).

```
Rule REQUEST_HEADERS:User-Agent @pm "one two three"
```

If the `capture` modifier is specified on a `@pm` rule, the `CAPTURE:0` variable will contain the matched data fragment. Do note that, because the `pm` operator can easily match many times per rule, the `CAPTURE:0` value is valid only when used in the same rule. In the following rules, `CAPTURE:0` will contain the data fragment of the last `@pm` match.

> **Note**
>
> DEPRECATED: The "ac" module is deprecated. Use `rx`, `dfa`, `match`, `imatch` or `ee_match_any` instead.

pmf

Description: Parallel matching with patterns from file.

Syntax:`@pmf filename`

Types: String

Module: ac

Version: 0.2

Same as `pm`, but instead of accepting parameters directly, it loads them from the file whose filename was supplied. The file is expected to contain one pattern per line. To convert a line into a pattern, whitespace from the beginning and the end is removed. Empty lines are ignored, as are comments, which are lines that begin with `#`. Relative filenames are resolved from same directory as the configuration file.

```
Rule REQUEST_HEADERS:User-Agent @pmf bad_user_agents.dat
```

> **Note**
>
> DEPRECATED: The "ac" module is deprecated. Use `rx`, `dfa`, `match`, `imatch` or `ee_match_any` instead.

rx

Description: Regular expression (perl compatible regular expression) matching.

Syntax:`@rx "pcre-regex"`

Types: String

Module: pcre

Version: 0.2

The `rx` operator implements PCRE [http://www.pcre.org/] regular expressions. The parameter of the operator is a regular expression pattern that is passed to the PCRE library without modification.

```
Rule ARGS:userId !@rx "^[0-9]+$"
```

Patterns are compiled with the following settings:

- Entire input is treated as a single buffer against which matching is done.
- Patterns are case-sensitive by default.
- Patterns are compiled with `PCRE_DOTALL` and `PCRE_DOLLAR_ENDONLY` set.

Using captured substrings to create variables

Regular expressions can be used to capture substrings. In IronBee, the captured substrings can be used to create new variables in the TX collection. To use this feature, specify the `capture` modifier in the rule.

```
Rule ARGS @rx "test(\d{13,16})" capture
```

When capture is enabled, IronBee will always create a variable CAPTURE:0, which will contain the entire matching area of the pattern. Anonymous capture groups will create up to 9 variables, from CAPTURE:1 to CAPTURE:9. These special CAPTURE variables will remain available until the next capture rule is run, when they will all be deleted.

streq

Description: Returns true if target exactly matches the given string.

Syntax:`@streq *"string"*`

Types: String

Module: core

Version: 0.3

istreq

Description: As `streq`, but case insensitive.

Syntax:`@istreq *"string"*`

Types: String

Module: core

Version: 0.7

Modifiers

...

allow

Description: Mark a transaction as allowed to proceed to a given inspection point.

Type: Action

Syntax: `allow[":phase"|":request"]`

Cardinality: 0..1

Module: core

Version: 0.4

By default this allows the transaction to proceed without inspection until the post-processing phase. This can be changed depending on the modifier used:

- **phase** - Proceed to the end of the current phase without further rule execution.
- **request** - Proceed to the end of the request processing phases without further rule execution.

event

Description: Cause the rule to generate a log event.

Type: Action

Syntax: `event[":observation"|":alert"]`

Cardinality: 0..1

Module: core

Version: 0.4

By default this generates a log event of type "observation", but this can be changed to type "alert". Having at least one active alert type event will cause an audit log to be generated.

- **observation** - Default event type denoting a rule made an observation, which could contribute to further inspection.
- **alert** - Alert event type denoting a transaction should be logged.

logdata

Description: Add data to be logged with the event.

Type: Metadata

Syntax: `logdata:`*`data`*

Cardinality: 0..1

Module: core

Version: 0.2

Log a data fragment as part of the error message.

```
Rule ARGS @rx pattern \
        "msg:Test matched" logdata:%{MATCHED_VAR}
```

Note: Up to 128 bytes of data will be recorded.

block

Description: Mark a transaction to be blocked.

Type: Action

Syntax: `block[:advisory | :phase | :immediate]`

Cardinality: 0..1

Module: core

Version: 0.4

By default this marks the transaction with an advisory blocking flag. This can be changed depending on the modifier used:

- **advisory** - Mark the transaction with an advisory blocking flag which further rules may take into account.
- **phase** - Block the transaction at the end of the current phase.
- **immediate** - Block the transaction immediately after rule execution.

capture

Description: Enable capturing the matching data.

Type: Modifier

Syntax: `capture`

Cardinality: 0..1

Module: core

Version: 0.4

Enabling capturing will populate the `CAPTURE` collection with data from the most recent matching operator. For most operators the `CAPTURE:0` field will be set to the last matching value. Operators that support capturing multiple values may set other items in the `CAPTURE` collection. For example, the `rx` operator supports setting the additional `CAPTURE:1` - `CAPTURE:9` via capturing parens in the regular expression and the `dfa` operator supports capturing *all matches*, each being available as `CAPTURE:0`.

chain

Description: Chains the next rule, so that the next rule will execute only if the current operator evaluates true.

Type: Modifier

Syntax: `chain`

Cardinality: 0..1

Module: core

Version: 0.4

Rule chains are essentially rules that are bound together by a logical AND with short circuiting. In a rule chain, each rule in the chain is executed in turn as long as the operators are evaluating true. If an operator evaluates to false, then no further rules in the chain will execute. This allows a rule to execute multiple operators.

All rules in the chain will still execute their actions before the next rule in the chain executes. If you want a rule that only executes an action if all operators evaluate true, then the action should be given on the final rule in the chain.

Requirements for chained rules:

- Only the first rule in the chain may have an id or phase, which will be used for all rule chains.
- A numeric chain ID will be assigned and appended to the rule ID, prefixed with a dash, to uniquely identify the rule.
- Different metadata attributes (except id/phase) may be given for each chain, but the first rule's metasta will be the default.
- Specifying one or more tag modifiers is allowed in any chain, but the tags will be bound to the entire rule chain so that RuleEnable and similar will act on the entire rule chain, not just an individual rule in the chain.

Example:

```
# Start a rule chain, which matches only POST requests. The implicit ID here
# will be set to "id:1-1".
Rule REQUEST_METHOD "@rx ^(?i:post)$" id:1 phase:REQUEST chain

# Only if the above rule's operator evaluates true, will the next rule in the
# chain execute. This rule checks to see if there are any URI based parameters
# which typically should not be there for POST requests. If the operator evaluates
# true, then the setvar action will execute, marking the transaction and an
# event will be generated with the given msg text. This rule will have the
# implicit ID set to "id:1-2".
Rule &REQUEST_URI_PARAMS @gt 0 "msg:POST with URI parameters." setvar:TX:uri_params_in_post=1 event

# Only if the above two rules' operators return true will the next rule in the
# chain execute.  This rule checks that certain parameters are not used in
# on the URI and if so, generates an event and blocks the transaction with the
# default status code at the end of the phase. This rule will have the implicit
# ID set to "id:1-3".
Rule &REQUEST_URI_PARAMS:/^(id|sess)$/ @gt 0 "msg:Sensitive parameters in URI." event block:phase
```

confidence

Description: Numeric value indicating the confidence of the rule.

Type: Metadata

Syntax: `confidence:`*`integer (0-100)`*

Cardinality: 0..1

Module: core

Version: 0.4

Higher confidence rules should have a lower False Positive rate.

delRequestHeader

Description: Delete an HTTP header from the request.

Type: Action

Syntax: `delRequestHeader:`*`header-name`*

Cardinality: 0..n

Module: core

Version: 0.4

delResponseHeader

Description: Delete an HTTP header from the response.

Type: Action

Syntax: `delResponseHeader:`*`header-name`*

Cardinality: 0..n

Module: core

Version: 0.4

id

Description: Unique identifier for a rule.

Type: Metadata

Syntax: `id:`*`name`*

Cardinality: 1

Module: core

Version: 0.4

Specifies a unique identifier for a rule. If a later rule re-uses the same identifier, then it will overwrite the previous rule.

TODO: Explain what the full unique id is (taking context and chains into account)

msg

Description: Message associated with the rule.

Type: Metadata

Syntax: `msg:`*`text`*

Cardinality: 0..1

Module: core

Version: 0.4

This message is used by the `event` action when logging the event.

phase

Description: The runtime phase at which the rule should execute.

Type: Metadata

Syntax: `phase:REQUEST_HEADER|REQUEST|RESPONSE_HEADER|RESPONSE|POSTPROCESS`

Cardinality: 1

Module: core

Version: 0.4

Rule phase determines when a rule runs. IronBee understands the following phases:

`REQUEST_HEADER`

: Invoked after the entire HTTP request headers has been read, but before reading the HTTP request body (if any). Most rules should not use this phase, opting for the `REQUEST` phase instead.

`REQUEST`

: Invoked after receiving the entire HTTP request, which may involve request body and request trailers, but it will run even when neither is present.

`RESPONSE_HEADER`

: Invoked after receiving the HTTP entire response header.

`RESPONSE`

: Invoked after receiving the HTTP response body (if any) and response trailers (if any).

`POSTPROCESS`

: Invoked after the entire transaction has been processed. This phase is for logging and tracking data between transactions, such as storing state. Actions cannot affect the transaction in this phase.

rev

Description: An integer rule revision.

Type: Metadata

Syntax: `rev:`*`integer (1-n)`*

Cardinality: 0..1

Module: core

Version: 0.4

TODO: Explain how this is used in RuleEnable and when overriding Rules in sub contexts.

setflag

Description: Set, or unset, boolean transaction attributes (flags).

Type: Action

Syntax: `setflag:[!]flag-name`

Cardinality: 0..n

Module: core

Version: 0.6

Allow setting or unsetting transaction flags. Prefixing with a `!` unsets the flag.

> **Note**
>
> Currently the `inspectRequestHeader` flag is always set as this is required for the site selection process. Additionally, the `RequestBuffering` and `ResponseBuffering` directives must be enabled to buffer the request or response.

- **block** - Set if transaction was marked for block.
- **suspicious** - Set if transaction was marked as suspicious and care should be taken in processing.
- **inspectRequestHeader** - Set if the engine should inspect the HTTP request header (default: set).
- **inspectRequestBody** - Set if the engine should inspect the HTTP request body (default: unset).
- **inspectResponseHeader** - Set if the engine should inspect the HTTP response header (default: unset).
- **inspectResponseBody** - Set if the engine should inspect the HTTP response body (default: unset).

setRequestHeader

Description: Set the value of a HTTP request header.

Type: Action

Syntax: `setRequestHeader:header-name=header-value`

Cardinality: 0..n

Module: core

Version: 0.4

setResponseHeader

Description: Set the value of an HTTP response header.

Type: Action

Syntax: `setResponseHeader:header-name=header-value`

Cardinality: 0..n

Module: core
Version: 0.4

setvar

Description: Set a variable data field.
Type: Action
Syntax: `setvar:[!]variable-field-name[+|-]=value`
Cardinality: 0..n
Module: core
Version: 0.2
The `setvar` modifier is used for data field manipulation. To create a variable data field or change its value:

```
setvar:tx:score=1
```

To remove all instances of a named variable data field:

```
setvar:!tx:score
```

To increment or decrement a variable data field value:

```
setvar:tx:score+=5
    setvar:tx:score-=5
```

An attempt to modify a value of a non-numerical variable will assume the old value was zero (NOTE: Probably should just fail, logging an attempt was made to modify a non-numerical value).

severity

Description: Numeric value indicating the severity of the issue this rule is trying to protect against.
Type: Metadata
Syntax: `severity:integer (0-100)`
Cardinality: 0..1
Module: core
Version: 0.4
The severity indicates how much impact a successful attack may be, but does not indicate the quality of protection this rule may provide. The severity is meant to be used as part of a "threat level" indicator. The "threat level" is essentially severity x confidence, which balances how severe the threat may be with how well this rule might be protecting against it.

status

Description: The HTTP status code to use for a blocking action.

Type: Modifier

Syntax: `status:`*`http-status-code`*

Cardinality: 0..1

Module: core

Version: 0.4

t

Description: Apply one or more named transformations to each of the targets in a rule.

Type: Modifier

Syntax: `t:`*`transformation-name`*

Cardinality: 0..n

Module: core

Version: 0.4

tag

Description: Apply an arbitrary tag name to a rule.

Type: Metadata

Syntax: `tag:`*`name`*

Cardinality: 0..n

Module: core

Version: 0.4

TODO: Describe where this is used, notably `RuleEnable`/`RuleDisable` and logged with events.

Transformation Functions

...

base64Decode

Description:

Module: core

Version: Not implemented yet.

compressWhitespace

Description: Replaces one or more consecutive whitespace characters with a single space.

Module: core

Version: 0.3

Replaces various whitespace characters with spaces. In addition, consecutive whitespace characters will be reduced down to a single space. Whitespace characters are: `0x20`, `\f`, `\t`, `\n`, `\r`, `\v`, `0xa0` (non-breaking whitespace).

count

Description: Given a collection, it returns the number if items in the collection. Given a scalar, returns 1.

Module: core

Version: 0.4

htmlEntityDecode

Description: Decodes HTML entities in the data.

Module: core

Version: 0.6

The following forms are supported:

- **&#*DDDD*;** - Numeric code point, where *DDDD* represents a decimal number with any number of digits.
- **&#x*HHHH*;** - Numeric code point, where HHHH represents a hexadecimal number with any number of digits.
- **&*name*;** - Predefined XML named entities (currently: quot, amp, apos, lt, gt).

See `https://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references`.

length

Description: Returns the byte length of the value.

Module: core

Version: 0.4

lowercase

Description: Returns the input as all lower case characters.

Module: core

Version: 0.2

removeWhitespace

Description: Removes one or more consecutive whitespace characters.

Module: core

Version: 0.3

Similar to `compressWhitespace`, except removes the characters instead of replacing them with a single space.

removeComments

Description: Remove various types of code comments.

Module: core

Version: Not implemented yet.

The following style comments are replaced:

- **/* ... */** - C style comments.
- **// ...** - C++ style comments.
- **# ...** - Shell style comments.
- **-- ...** - SQL style comments.

replaceComments

Description: Replace various types of code comments with a single space character.

Module: core

Version: Not implemented yet.

This is similar to `removeComments`, but instead of removing, replaces with a single space character.

trim

Description: Removes consecutive whitespace from the beginning and end of the input.

Module: core

Version: 0.2

trimLeft

Description: Removes consecutive whitespace from the beginning of the input.

Module: core

Version: 0.2

trimRight

Description: Removes consecutive whitespace from the end of the input.

Module: core

Version: 0.2

urlDecode

Description: Decodes URL encoded values in the input.

Module: core

Version: Not implemented yet.

Implements decoding the encoding used in application/x-www-form-urlencoded values (percent encoding with additions).

- **%*HH*;** - Numeric code point, where HH represents a two digit hexadecimal number.
- **+** - Represents an ASCII space character (equiv to `%20`).

> **Warning**
>
> Fields which are parsed from the URI and form parameters are already URL Decoded and you should not apply this transformation to these fields unless you are trying to inspect multiple levels of encoding.

min

Description: Given a collection of numeric data, returns the minimum value.

Module: core

Version: 0.3

max

Description: Given a collection of numeric data, returns the maximum value.

Module: core

Version: 0.3

normalizePath

Description: Normalize a filesystem path, removing back and self references.

Module: core

Version: 0.6

normalizePathWin

Description: Normalize a Windows filesystem path, removing back and self references.

Module: core

Version: 0.6

normalizeSqli

Description: Normalize potential SQL injection via libinjection.

Module: libinjection

Version: 0.7

The libinjection ironbee module utilizes Nick Galbreath's libinjection to implement SQLi detection. This transformation is based on an example in libinjection. The libinjection library is available via: http://www.client9.com/projects/libinjection/

Example Input/Output:

```
Input: foo' /* x */   or 1/* y -- */=/* z */1 union select id,passwd from users --
```

```
Output: foo' or 1=1 union select id,passwd from users --
```

normalizeSqlPg

Description: Normalize postgres SQL.

Module: sqltfn

Version: 0.7

Normalize Postgres SQL.

Example Input/Output:

```
Input: foo' /* x */   or 1/* y -- */=/* z */1 union select id,passwd from users --
```

```
Output: foo' or 1 = 1 union select id,passwd from users
```

Chapter 7: Extending IronBee

...

Overview

...

> **Warning**
>
> This documentation is currently out of date.

Execution Flow

Definitions

Engine
: The framework that controls data flow, state and code execution.

Server Plugin
: Server native code for embedding the engine into another software base (e.g. the Apache httpd server). The plugin is responsible for instantiating the engine, initiating the initial configuration process, feeding the engine with data and optionally implementing methods of blocking.

Hook
: A hook is an execution point within the engine that allows external code to be registered and executed as if it were part of the engine. There are many builtin hooks in the IronBee engine and custom hooks can also be added. Hooks are typically leveraged by modules.

Module
: Engine code that is not essential to the core engine, but rather extends what the engine can accomplish by hooking into it. Modules in IronBee are dynamically loadable files which can extend and alter how the engine executes. There are a number of different types of modules which will be explained in detail. Some examples of modules are HTTP parsers, matching algorithms, logging methods, rule languages/executors and specialized detection techniques. All IronBee features are essentially modules, which allows nearly every aspect of the engine to be extended.

Flow

There are four main stages of execution detailed below.

Startup Stage

During startup, the plugin is instantiated by whatever server has loaded it, for example when the Apache httpd server loads/configures the plugin. During this stage, the plugin instantiates the engine and initiates the configuration stage.

1. Server starts and instantiates/starts the plugin.
2. Server Plugin is configured with native plugin configuration, which includes the location of the engine configuration.
3. Engine is instantiated.
   a. An engine configuration context is created.
   b. Static core module is loaded.
4. Server Plugin registers a native logging provider.
5. Engine configuration stage is initiated based on initial plugin configuration.

Configuration Stage

During configuration, the configuration files/scripts are read/executed, engine modules are loaded/initialized and contexts are created/configured in preparation for the runtime stage. The following is an outline of what will happen during this stage.

1. Configuration is read/executed.
2. The main configuration context is created.
3. Modules are loaded.
   a. Module global configuration data is copied to the global context as a base configuration.
   b. Module "init" function is called just after it is loaded to initialize any module configuration.
   c. Modules may hook into the engine by registering to be called when certain events occur.
   d. If successfully initialized, a module is registered with the engine.
4. Configuration contexts are created and registered.
5. Modules register themselves with a configuration context if they are to be used in that context.
   a. Module "context init" function is called to initialize any context configuration.
   b. Modules may hook into the engine for the given context by registering to be called when certain events occur.
6. The runtime stage is initiated.

Runtime Stage

During runtime all of the configuration has been finalized and the engine will now handle data passed to it by the plugin. Data is handled by the state machine which essentially follows a five step process. First, a configuration context is chosen. Second, the request is handled. Third the response is handled. Forth, any post processing is performed. And finally, logging is performed. Below is an outline of the flow.

TODO: Below is no longer true. Rewrite based on removal of unparsed data interface.

1. Raw connection HTTP data is received by the server plugin and passed to the engine.
2. [Need to add connection context here. Events could be: conn open, conn data (inbound/outbound), conn close. Configuration options include which protocol parser to use, default parser configuration, whether to decrypt SSL, private keys for decryption, etc.]
3. If the connection is encrypted, SSL decryption takes place. This step is optional and will largely depend on how the plugin is designed. For example, the Apache plugin will always send decrypted data.
4. The engine parses the data as a stream, buffering if configured to do so.
5. The parser notifies the engine of various events (request headers available, request body, etc.)
6. Any hooks associated with events are executed.
7. Once enough data is available, the configuration context selection process is started configuration context function until one returns that it wants to be enabled.
   a. At this point all modules registered in the chosen context will have their "context activated" functions executed, allowing them to be prepared for executing in the context.
8. Further events occur and associated hooks are executed, but now with the chosen configuration context instead of the global context.

Hooks

TODO: Add description of each hook

Modules

Modules make up the majority of executed code in IronBee. Most features are built using modules. There are three primary reasons for this. First, it makes the code more readable and each feature more self contained. Second, it allows only features in use to be loaded into the executable. And last, since modules are shared libraries, it makes for easier upgrades as the engine only needs to unload the old code and reload the new.

Modules have three essential duties. A module must export a known symbol so that it can be loaded. A set of configuration parameters may be set. And common module functions must be

registered which will be called at various initialization and cleanup points. With Lua, however, this is much more simplified than in C.

Exporting a symbol is quite language specific and will not be discussed here.

Any number of configuration parameters are registered with the engine and their storage locations are then mapped by the engine both globally to the module as well as into each configuration context. As of this writing, there are two types of configuration parameters, numeric and string. Along with configuration parameter definitions can be defined default values.

The eventual goal of a module is to register functions to be called by the engine. This is done by registering callback functions to be called with hooks. Hooks allow executing at defined points in the connection/transaction lifecycle, which is documented with the state machine in the API documentation.

TODO: Need more on what a basic module will look like without going into language details.

Writing Modules in C

TODO: Some general description on why one would want to do this.

Anatomy of a C Module

A C module is built into a shared library. The shared library exposes a known structure (see Example 7.1) that IronBee uses to load the module.

Example 7.1. IronBee Module Structure

```
struct ib_module_t {
    /* Header */
    uint32_t                   vernum;           /* Engine version number */
    uint32_t                   abinum;           /* Engine ABI Number */
    const char                *version;          /* Engine version string */
    const char                *filename;         /* Module code filename */
    void                      *data;             /* Module data */
    ib_engine_t               *ib;               /* Engine */
    ib_rule_t                 *rule;             /* Module Rule. */
    size_t                     idx;              /* Module index */

    /* Module Config */
    const char                *name;             /* Module name */
    const void                *gcdata;           /* Global config data */
    size_t                     gclen;            /* Global config data length */
    ib_module_fn_cfg_copy_t    fn_cfg_copy;      /* Config copy handler */
    void                      *cbdata_cfg_copy;  /* Config copy data */
    const ib_cfgmap_init_t    *cm_init;          /* Module config mapping */
    const ib_dirmap_init_t    *dm_init;          /* Module directive mapping */

    /* Functions */
    ib_module_fn_init_t        fn_init;          /* Module init */
    void                      *cbdata_init;      /*  fn_init callback data */
    ib_module_fn_fini_t        fn_fini;          /* Module finish */
    void                      *cbdata_fini;      /*  fn_fini callback data */

    /* DEPRECATED: Will move to hooks */
    ib_module_fn_ctx_open_t    fn_ctx_open;      /* Context open */
    void                      *cbdata_ctx_open;  /*  fn_ctx_open callback data */
    ib_module_fn_ctx_close_t   fn_ctx_close;     /* Context close */
    void                      *cbdata_ctx_close; /*  fn_ctx_close callback data */
    ib_module_fn_ctx_destroy_t fn_ctx_destroy;   /* Context destroy */
    void                      *cbdata_ctx_destroy;/*  fn_ctx_destroy callback data */
};
```

A module must define and initialize this structure to be loadable in IronBee. This is done by defining a few functions and making a few macro calls. A minimal module example is given in Example 7.2.

Example 7.2. Minimal Module

```
#include <ironbee/cfgmap.h>
#include <ironbee/engine.h>
#include <ironbee/module.h>


/* Declare the public module symbol. */
IB_MODULE_DECLARE();

/* Called when module is loaded. */
static ib_status_t exmin_init(ib_engine_t *ib, ib_module_t *m, void *cbdata)
{
    ib_log_debug(ib, 4, "Example minimal module loaded.");
    return IB_OK;
}

/* Called when module is unloaded. */
static ib_status_t exmin_fini(ib_engine_t *ib, ib_module_t *m, void *cbdata)
{
    ib_log_debug(ib, 4, "Example minimal module unloaded.");
    return IB_OK;
}

/* Initialize the module structure. */
IB_MODULE_INIT(
    IB_MODULE_HEADER_DEFAULTS,      /* Default metadata */
    "exmin",                        /* Module name */
    IB_MODULE_CONFIG_NULL,          /* Global config data */
    NULL,                           /* Configuration field map */
    NULL,                           /* Config directive map */
    exmin_init,                          /**< Initialize function */
    NULL,                                /**< Callback data */
    exmin_fini,                          /**< Finish function */
    NULL,                                /**< Callback data */
    NULL,                                /**< Context open function */
    NULL,                                /**< Callback data */
    NULL,                                /**< Context close function */
    NULL,                                /**< Callback data */
    NULL,                                /**< Context destroy function */
    NULL                                 /**< Callback data */
);
```

Example 7.2 shows a very minimalistic module that does nothing but log when the module loads and unloads. The module includes some standard IronBee headers, declares itself a module and defines two functions. The module structure is then initialized with these functions assigned to the `fn_init` and `fn_fini` fields. This results in the `exmin_init` and `exmin_fini` functions being called when the module is loaded and unloaded, respectfully. Of course much more can be done with a module.

TODO: Describe what other things a module can do.

A Simple C Module Example

TODO: This example is outdated and needs to be updated/replaced.

To better illustrate writing a C module we need a simple task to accomplish. Here we will define a minimalistic signature language. To keep things simple, the module will stick to IronBee built-in features and ignore any performance concerns. The module will simply allow a user to add signature to IronBee. In this case a signature is defined as performing a PCRE based regular expression on a given data field and triggering an event if there is a match.

To accomplish this task, we need to write a module that does the following:

- Allow writing a signature within the configuration file that allows specifying when it should execute, what field it should match against, a regular expression and an event message that should be triggered on match.
- Parse the signature into its various components.
- Compile the PCRE and store the signature for later execution.
- At runtime, execute the signatures at the specified time.
- If a signature matches, generate an event.

The module begins the same as in Example 7.2, but with some additional type definitions which we will use to store our signatures.

Example 7.3. Signature Module Setup

```
#include <strings.h>

#include <ironbee/engine.h>
#include <ironbee/debug.h>
#include <ironbee/mpool.h>
#include <ironbee/cfgmap.h>
#include <ironbee/module.h>
#include <ironbee/provider.h>

/* Define the module name as well as a string version of it. */
#define MODULE_NAME               pocsig
#define MODULE_NAME_STR           IB_XSTRINGIFY(MODULE_NAME)

/* Declare the public module symbol. */
IB_MODULE_DECLARE();

typedef struct pocsig_cfg_t pocsig_cfg_t;
typedef struct pocsig_sig_t pocsig_sig_t;

/* Signature Phases */
typedef enum {
    POCSIG_PRE,                   /* Pre transaction phase */
    POCSIG_REQHEAD,               /* Request headers phase */
    POCSIG_REQ,                   /* Request phase */
    POCSIG_RESHEAD,               /* Response headers phase */
    POCSIG_RES,                   /* Response phase */
    POCSIG_POST,                  /* Post transaction phase */

    /* Keep track of the number of defined phases. */
    POCSIG_PHASE_NUM
} pocsig_phase_t;

/* Signature Structure */
struct pocsig_sig_t {
    const char         *target;   /* Target name */
    const char         *patt;     /* Pattern to match in target */
    void               *cpatt;    /* Compiled PCRE regex */
    const char         *emsg;     /* Event message */
};
```

Configuration

Modules control their own configuration structure. Normally a module will use a simple C structure which it can reference directly. However, a module may also expose some or all of its configuration. Any exposed parameters can then be accessed by other modules and/or through the configuration language. In addition to exposing configuration parameters a module can register and expose new configuration directives for use in the configuration language.

In this example we will need to track multiple lists of signatures (one for each point of execution) and a handle to the PCRE pattern matcher. While these will not be exposed, we will expose a numeric parameter to toggle tracing signature execution. The configuration is defined and instantiated in a C structure shown in Example 7.4.

Example 7.4. Configuration Structure

```
/* Module Configuration Structure */
struct pocsig_cfg_t {
    /* Exposed as configuration parameters. */
    ib_num_t            trace;                  /* Log signature tracing */

    /* Private. */
    ib_list_t          *phase[POCSIG_PHASE_NUM]; /* Phase signature lists */
    ib_matcher_t       *pcre;                   /* PCRE matcher */
};

/* Instantiate a module global configuration. */
static pocsig_cfg_t pocsig_global_cfg;
```

We will then define a configuration directive to control tracing as well as signature directives for each phase of execution. Note that multiple signature directives are only used to simplify the example so that we do not have to write rule parsing code. The functions defined in Example 7.5 are used to handle the configuration directives, which we will define later on.

The `pocsig_dir_trace` function is a simple single parameter directive handler which parses the parameter for a "On" or "Off" value and sets a numeric parameter value in the configuration context. We will see how this parameter is exposed later on. The `pocsig_dir_signature` function is a directive handler that can handle an arbitrary number of parameters. Note that much of this function is described later on with pattern matchers.

```
    if (rc != IB_OK) {
        ib_log_error(ib, 1, "No PocSig target");
        return IB_EINVAL;
    }

    /* Operator */
    rc = ib_list_shift(args, &op);
    if (rc != IB_OK) {
        ib_log_error(ib, 1, "No PocSig operator");
        return IB_EINVAL;
    }

    /* Action */
    rc = ib_list_shift(args, &action);
    if (rc != IB_OK) {
        ib_log_debug(ib, 4, "No PocSig action");
        action = "";
    }

    /* Signature */
    sig = (pocsig_sig_t *)ib_mpool_alloc(ib_engine_pool_config_get(ib),
                                         sizeof(*sig));
    if (sig == NULL) {
        return IB_EALLOC;
    }

    sig->target = ib_mpool_memdup(ib_engine_pool_config_get(ib),
                                  target, strlen(target));
    sig->patt = ib_mpool_memdup(ib_engine_pool_config_get(ib),
                                op, strlen(op));
    sig->emsg = ib_mpool_memdup(ib_engine_pool_config_get(ib),
                                action, strlen(action));

    /* Compile the PCRE patt. */
    if (cfg->pcre == NULL) {
        ib_log_error(ib, 2, "No PCRE matcher available (load the pcre module?)");
        return IB_EINVAL;
    }
    sig->cpatt = ib_matcher_compile(cfg->pcre, sig->patt, &errptr, &erroff);
    if (sig->cpatt == NULL) {
        ib_log_error(ib, 2, "Error at offset=%d of PCRE patt=\"%s\": %s",
                     erroff, sig->patt, errptr);
        return IB_EINVAL;
    }

    ib_log_debug(ib, 4, "POCSIG: \"%s\" \"%s\" \"%s\" phase=%d ctx=%p",
                 target, op, action, phase, ctx);

    /* Add the signature to the phase list. */
    rc = ib_list_push(cfg->phase[phase], sig);
    if (rc != IB_OK) {
        ib_log_error(ib, 1, "Failed to add signature");
        return rc;
    }

    return IB_OK;
}
```

Any configuration parameters and directives must be registered with the engine. This is accomplished through two mapping structures as shown in Example 7.6. The exposed configuration parameter is named, typically modulename.name, and the engine told it type, offset, length and default value. This is wrapped into a macro to make this much easier. The configuration directives are registered in a similar fashion and mapped to handler functions.

```
    /* trace */
    IB_CFGMAP_INIT_ENTRY(
        MODULE_NAME_STR ".trace",
        IB_FTYPE_NUM,
        pocsig_cfg_t,
        trace,
        0
    ),

    /* End */
    IB_CFGMAP_INIT_LAST
};

/* Directive initialization structure. */
static IB_DIRMAP_INIT_STRUCTURE(pocsig_directive_map) = {
    /* PocSigTrace - Enable/Disable tracing */
    IB_DIRMAP_INIT_PARAM1(
        "PocSigTrace",
        pocsig_dir_trace,
        NULL
    ),

    /* PocSig* - Define a signature in various phases */
    IB_DIRMAP_INIT_LIST(
        "PocSigPreTx",
        pocsig_dir_signature,
        NULL
    ),
    IB_DIRMAP_INIT_LIST(
        "PocSigReqHead",
        pocsig_dir_signature,
        NULL
    ),
    IB_DIRMAP_INIT_LIST(
        "PocSigReq",
        pocsig_dir_signature,
        NULL
    ),
    IB_DIRMAP_INIT_LIST(
        "PocSigResHead",
        pocsig_dir_signature,
        NULL
    ),
    IB_DIRMAP_INIT_LIST(
        "PocSigRes",
        pocsig_dir_signature,
        NULL
    ),
    IB_DIRMAP_INIT_LIST(
        "PocSigPostTx",
        pocsig_dir_signature,
        NULL
    ),

    /* End */
    IB_DIRMAP_INIT_LAST
};
```

Pattern Matchers

Pattern matchers are defined through the matcher provider interface. These matchers are typically loaded via modules. In case of the PCRE matcher, it is loaded through the pcre module, which must be loaded for our example module to work. A matcher provider exposes a common interface for calling any pattern matchers registered with the engine.

In Example 7.5 `ib_matcher_create` is used to fetch the PCRE pattern matcher. This matcher is used here to compile the patterns with ib_matcher_compile. The matcher is stored in the configuration context for later use in executing the signatures. The compiled pattern is stored in the signature structure which is added to a list for later execution.

Hooks

Up until now, we have been dealing with configuration time processing. In order to handle processing at runtime, we have to define a handler and register this handler to be executed at defined points. Since all signatures are executed in the same fashion, we can define a single handler and register it to be executed multiple times.

```
    }

    ib_log_debug(ib, dbglvl, "Executing %d signatures for phase=%d ctx=%p",
                 ib_list_elements(sigs), phase, tx->ctx);

    /* Run all the sigs for this phase. */
    IB_LIST_LOOP(sigs, node) {
        pocsig_sig_t *s = (pocsig_sig_t *)ib_list_node_data(node);
        ib_field_t *f;

        /* Fetch the field. */
        rc = ib_data_get(tx->dpi, s->target, &f);
        if (rc != IB_OK) {
            ib_log_error(ib, 4, "PocSig: No field named \"%s\"", s->target);
            continue;
        }

        /* Perform the match. */
        ib_log_debug(ib, dbglvl, "PocSig: Matching \"%s\" against field \"%s\"",
                     s->patt, s->target);
        rc = ib_matcher_match_field(cfg->pcre, s->cpatt, 0, f, NULL);
        if (rc == IB_OK) {
            ib_logevent_t *e;

            ib_log_debug(ib, dbglvl, "PocSig MATCH: %s at %s", s->patt, s->target);

            /* Create the event. */
            rc = ib_logevent_create(
                &e,
                tx->mp,
                "-",
                IB_LEVENT_TYPE_ALERT,
                IB_LEVENT_ACT_UNKNOWN,
                IB_LEVENT_PCLASS_UNKNOWN,
                IB_LEVENT_SCLASS_UNKNOWN,
                IB_LEVENT_SYS_UNKNOWN,
                IB_LEVENT_ACTION_IGNORE,
                IB_LEVENT_ACTION_IGNORE,
                90,
                80,
                s->emsg
            );
            if (rc != IB_OK) {
                ib_log_error(ib, 3, "PocSig: Error generating event: %d", rc);
                continue;
            }

            /* Log the event. */
            ib_event_add(tx->epi, e);
        }
        else {
            ib_log_debug(ib, dbglvl, "PocSig NOMATCH");
        }
    }

    return IB_OK;
}
```

Example 7.7 defines a handler for executing our signatures at runtime. In order to use this handler with each phase, we will pass the phase number to the handler. Other than some casting trickery to pass the phase number, the function is fairly straight forward. It loops through a phase list, fetches the data field it will match against, matches the pre-compiled pattern against the field and then logs an event if there is a match.

All that is left in the module is to register the signature handler to be executed in the various phases. Example 7.8 shows the finial module functions and registration required for this. Normally configuration data is exposed publicly where it is given a default value. Since some of our configuration is not exposed, we need to initialize the data ourselves. This is done though the module initialization function, `pocsig_init`. The context initialization function, `pocsig_context_init`, is called for each configuration context that this module is configured. This is where we register our handler with the engine hooks and define the phase numbers that are passed to the handler. Finally, the module structure is initialized to point to the various configuration mapping structures and module initialization functions.

Example 7.8. Module Functions and Registration

```
static ib_status_t pocsig_init(ib_engine_t *ib,
                               ib_module_t *m)
{

    /* Register hooks to handle the phases. */
    ib_hook_tx_register(ib, handle_context_tx_event,
                        pocsig_handle_sigs, (void *)POCSIG_PRE);
    ib_hook_tx_register(ib, handle_request_headers_event,
                        pocsig_handle_sigs, (void *)POCSIG_REQHEAD);
    ib_hook_tx_register(ib, handle_request_event,
                        pocsig_handle_sigs, (void *)POCSIG_REQ);
    ib_hook_tx_register(ib, handle_response_headers_event,
                        pocsig_handle_sigs, (void *)POCSIG_RESHEAD);
    ib_hook_tx_register(ib, handle_response_event,
                        pocsig_handle_sigs, (void *)POCSIG_RES);
    ib_hook_tx_register(ib, handle_postprocess_event,
                        pocsig_handle_sigs, (void *)POCSIG_POST);

    return IB_OK;
}

IB_MODULE_INIT(
    IB_MODULE_HEADER_DEFAULTS,           /* Default metadata */
    MODULE_NAME_STR,                     /* Module name */
    IB_MODULE_CONFIG(&pocsig_global_cfg),/* Global config data */
    pocsig_config_map,                   /* Configuration field map */
    pocsig_directive_map,                /* Config directive map */
    pocsig_init,                         /* Initialize function */
    NULL,                                /* Finish function */
    NULL,                                /* Context init function */
    NULL                                 /* Context fini function */
);
```

Events

TODO

Writing Modules in Lua

Lua modules are designed to be much easier to develop than a C equivalent. A Lua IronBee module is built like any other Lua module. Really all you need to do is to implement handlers which are registered to execute when an event is triggered.

```
-- ====================================================================
-- ====================================================================
-- Licensed to Qualys, Inc. (QUALYS) under one or more
-- contributor license agreements.  See the NOTICE file distributed with
```

```
-- this work for additional information regarding copyright ownership.
-- QUALYS licenses this file to You under the Apache License, Version 2.0
-- (the "License"); you may not use this file except in compliance with
-- the License.  You may obtain a copy of the License at
--
--     http://www.apache.org/licenses/LICENSE-2.0
--
-- Unless required by applicable law or agreed to in writing, software
-- distributed under the License is distributed on an "AS IS" BASIS,
-- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
-- See the License for the specific language governing permissions and
-- limitations under the License.
-- ============================================================================
-- ============================================================================
--
-- This is an example IronBee lua module.  Essentially, this code is
-- executed on load, allowing the developer to register other functions
-- to get called when events fire at runtime.
--
-- This example just registers a logging function for most events as well
-- as a more complex function to execute when the request headers are
-- ready to process.
--
-- Author: Sam Baskinger <sbaskinger@qualys.com>
-- Author: Brian Rectanus <brectanus@qualys.com>
-- ============================================================================

-- ================================================
-- Get an IronBee module object, which is passed
-- into the loading module as a parameter.  This
-- is used to access the IronBee Lua API.
-- ================================================
local ibmod = ...

-- ================================================
-- Register an IronBee configuration file
-- directive that takes a single string parameter:
--
--   LuaExampleDirective <string>
-- ================================================
ibmod:register_param1_directive(
    "LuaExampleDirective",
    function(ib_module, module_config, name, param1)
        -- Log that we're configuring the module.
        ibmod:logInfo("Got directive %s=%s", name, param1)

        -- Configuration, in this case, is simply storing the string.
        module_config[name] = param1
    end
)
```

```
-- ================================================
-- Generic function to log an info message when
-- an event fires.
-- ================================================
local log_event = function(ib, event)
    ib:logInfo(
        "Handling event=%s: LuaExampleDirective=%s",
        ib.event_name,
        ib.config["LuaExampleDirective"])
    return 0
end

-- ================================================
-- This is called when a connection is started.
-- ================================================
ibmod:conn_started_event(log_event)

-- ================================================
-- This is called when a connection is opened.
-- ================================================
ibmod:conn_opened_event(log_event)

-- ================================================
-- This is called when a connection context was
-- chosen and is ready to be handled.
-- ================================================
ibmod:handle_context_conn_event(log_event)

-- ================================================
-- This is called when the connection is ready to
-- be handled.
-- ================================================
ibmod:handle_connect_event(log_event)

-- ================================================
-- This is called when the transaction starts.
-- ================================================
ibmod:tx_started_event(log_event)

-- ================================================
-- This is called when a request starts.
-- ================================================
ibmod:request_started_event(log_event)

-- ================================================
-- This is called when the transaction context
-- is ready to be handled.
-- ================================================
ibmod:handle_context_tx_event(log_event)

-- ================================================
```

```
-- This is called when there is new request
-- header data.
-- ================================================
ibmod:request_header_data_event(log_event)

-- ================================================
-- This is called when the request header data
-- has all been received.
-- ================================================
ibmod:request_header_finished_event(log_event)

-- ================================================
-- This is called when the request headers are
-- available to inspect.
-- ================================================
ibmod:handle_request_header_event(
    function(ib)
        log_event(ib)

        -- You can get named fields.  Scalar fields
        -- will return scalar values.
        local req_line = ib:get("request_line")
        ib:logInfo("REQUEST_LINE: %s=%s", type(req_line), tostring(req_line))

        -- You can fetch collections as a table of name/value pairs:
        local req_headers = ib:get("request_headers")
        if type(req_headers) == 'table' then
            -- Loop over the key/field
            for k,f in pairs(req_headers) do
                if type(f) == 'table' then
                    -- Fields come as (tables), which you can
                    -- unpack into simple name/value pairs.
                    --
                    -- TODO: Need to make this a bit cleaner
                    name, val = unpack(f)
                    ib:logInfo("REQUEST_HEADERS:%s=%s", tostring(name), tostring(val))
                else
                    ib:logInfo("REQUEST_HEADERS:%s=%s", k, f)
                end
            end
        end

        -- You can access individual subfields within collections directly
        -- via "name:subname" syntax, but these will come as a list
        -- of values (as more than one subname is always allowed):
        local http_host_header = ib:getValues("request_headers:host")
        ib:logInfo("First HTTP Host Header: %s", http_host_header[1])

        -- Request cookies are a collection (table of field objects)
        -- similar to headers:
        local req_cookies = ib:get("request_cookies")
```

```
        if type(req_cookies) == 'table' then
            -- Loop over the key/field
            for k,f in pairs(req_cookies) do
                if type(f) == 'table' then
                    -- Fields come as (tables), which you can
                    -- unpack into simple name/value pairs.
                    --
                    -- TODO: Need to make this a bit cleaner
                    name, val = unpack(f)
                    ib:logInfo("REQUEST_COOKIES:%s=%s", tostring(name), tostring(val))
                else
                    ib:logInfo("REQUEST_COOKIES:%s=%s", k, f)
                end
            end
        end

        return 0
    end
)

-- ================================================
-- This is called when the request body is
-- available.
-- ================================================
ibmod:request_body_data_event(log_event)

-- ================================================
-- This is called when the complete request is
-- ready to be handled.
-- ================================================
ibmod:handle_request_event(log_event)

-- ================================================
-- This is called when the request is finished.
-- ================================================
ibmod:request_finished_event(log_event)

-- ================================================
-- This is called when the transaction is ready
-- to be processed.
-- ================================================
ibmod:tx_process_event(log_event)

-- ================================================
-- This is called when the response is started.
-- ================================================
ibmod:response_started_event(log_event)

-- ================================================
-- This is called when the response headers are
-- available.
```

```
-- ================================================
ibmod:handle_response_header_event(log_event)

-- ================================================
-- This is called when the response headers are
-- ready to be handled.
-- ================================================
ibmod:response_header_data_event(log_event)

-- ================================================
-- This is called when the response header data
-- has all been received.
-- ================================================
ibmod:response_header_finished_event(log_event)

-- ================================================
-- This is called when the response body is
-- available.
-- ================================================
ibmod:response_body_data_event(log_event)

-- ================================================
-- This is called when the complete response is
-- ready to be handled.
-- ================================================
ibmod:handle_response_event(log_event)

-- ================================================
-- This is called when the response is finished.
-- ================================================
ibmod:response_finished_event(log_event)

-- ================================================
-- This is called after the transaction is done
-- and any post processing can be done.
-- ================================================
ibmod:handle_postprocess_event(log_event)

-- ================================================
-- This is called after postprocess is complete,
-- to allow for any post-transaction logging.
-- ================================================
ibmod:handle_logging_event(log_event)

-- ================================================
-- This is called when the transaction is
-- finished.
-- ================================================
ibmod:tx_finished_event(log_event)

-- ================================================
```

```
-- This is called when a connection is closed.
-- ================================================
ibmod:conn_closed_event(log_event)

-- ================================================
-- This is called when the connection disconnect
-- is ready to handle.
-- ================================================
ibmod:handle_disconnect_event(log_event)

-- ================================================
-- This is called when the connection is finished.
-- ================================================
ibmod:conn_finished_event(log_event)

-- ================================================
-- This is called when a logevent event has
-- occurred.
-- ================================================
ibmod:handle_logevent_event(log_event)

-- Report success.
ibmod:logInfo("Module loaded!")

-- Return IB_OK.
return 0
```

Chapter 8: Installation Guide

...

> **Note**
>
> This chapter is very much out-of-date and needs to be updated.

Tested Operating Systems

We have provided a table below of the operating systems that are officially supported by IronBee. Our definition of a tested operating system is one that we perform build, functionality, and regression testing on. This is not to say that IronBee will not work for you if your OS is not listed on this table, it probably will as long as you can meet the general dependencies outlined in the section "General Dependencies".

Table 8.1. Tested Operating Systems

Operating System	Version(s)	Website
Red Hat Enterprise Linux	Current and previous version.	http://www.redhat.com/rhel/
Fedora	Current version	http://fedoraproject.org/
Debian	Current stable version	http://www.debian.org/
Ubuntu-LTS	Current and previous version	https://wiki.ubuntu.com/LTS/
Ubuntu(non-LTS release)	Current version	http://www.ubuntu.com/
OS X	Lion	http://www.apple.com/

General Dependencies

...

Table 8.2. Build Tool Dependencies

Dependency	Version	Description	Website
C compiler	gcc 4.6+ or clang 3.0	Currently gcc and clang have been tested.	http://gcc.gnu.org/ or http://clang.llvm.org/
GNU Build System	autoconf 2.9+	Autotools(Automake, Autoconf, Libtool)	http://www.gnu.org/software/hello/manual/autoconf/The-GNU-Build-System.html

pkg-config	any	Helper tool used when compiling applications and libraries.	http://pkg-config.freedesktop.org/wiki/

Table 8.3. Software Version Control

Dependency	Version	Description	Website
Git	latest	Git is needed to access the IronBee source repository.	http://git-scm.com/

Table 8.4. Libraries for IronBee Engine

Dependency	Version	Description	Website
PCRE	8.0+	Regular Expression Library.	http://www.pcre.org/
PThread	NA	POSIX threads	NA
ossp-uuid	1.6.2+	OSSP UUID library.	http://www.ossp.org/pkg/lib/uuid/

Table 8.5. Libraries for IronBee C++ Wrapper and Utilities

Dependency	Version	Description	Website
C++ Compiler	g++ 4.6+ or clang++ 3.0	Currently gcc and clang have been tested.	http://gcc.gnu.org/ or http://clang.llvm.org/
Boost	1.46+	General purpose C++ library.	http://www.boost.org/

Table 8.6. Libraries for IronBee C++ CLI (clipp)

Dependency	Version	Description	Website
protobuf-cpp	2.4.1+	Generic serialization library.	https://developers.google.com/protocol-buffers/
libpcap	1.1.1+	Packet capture library (optional).	http://www.tcpdump.org/
libnids	latest	TCP reassembly library (optional).	http://libnids.sourceforge.net/
libnet	latest	Generic networking library (optional).	http://libnet.sourceforge.net/

stringencoders	3.10+	String encoder library (optional).	https://code.google.com/p/stringencoders/

Table 8.7. Server

Dependency	Version	Description	Website
Apache Traffic Server	3.1	Apache foundation's Traffic Server.	http://trafficserver.apache.org/

Building, Testing and Installing IronBee

...

Initial Setup

```
# Clone the repository
git clone git://github.com/ironbee/ironbee.git
cd ironbee

# Generate the autotools utilities
./autogen.sh
```

Build and Install IronBee Manually

IronBee is built using standard autotool conventions. First the source must be configured for the platform, then built and finally installed. Typically this is as simple as below, but there are many options to configure, which you can see by running the `./configure --help` script.

```
# Configure the build for the current platform
./configure

# Build
make

# Install (default is /usr/local/ironbee, but can be set with --prefix= option to configure above)
sudo make install
```

Build and Install IronBee as an RPM

Alternatively, you can build ironbee as an RPM.

```
# Configure the build enough to bootstrap the rpm-package build target
./configure
```

```
# Build the RPM
make rpm-package

# Install the RPMs (your architecture may differ)
sudo rpm -iv packaging/rpm/RPMS/x86_64/ironbee*.rpm
```

Build and Run Unit Tests(Optional)

IronBee comes with extensive unit tests. These are built and executed via the standard "check" make target:

```
make check
```

Build Doxygen Documents(Optional)

Developer (API) documentation is built into the IronBee source code. This can be rendered into HTML or PDF using the "doxygen" utility via the "doxygen" make target:

```
make doxygen
```

Build Docbook Manual(Optional)

The user manual is also part of IronBee. This is written in docbook 5 and currently requires a java runtime to build. This is built via the "manual" make target:

```
make manual
```

Appendix A: Configuration Examples

...

Example Trafficserver Configuration

```
# Insert this into trafficserver's plugin.config.
# Adjust paths as appropriate for your installation.

# First we need to load libraries the Ironbee plugin relies on.
/usr/local/ironbee/lib/libloader.so /usr/local/ironbee/lib/libironbee.so

# Now we can load the ironbee plugin.  The argument to this is ironbee's
# configuration file: see ironbee-config.txt
/usr/local/ironbee/lib/ts_ironbee.so /usr/local/trafficserver/etc/ironbee-ts.conf

#We could also use options to the ts_ironbee load line:
#  -l name   to specify a different filename for ironbee log messages.
#  -v n      to set the initial log level for Ironbee messages.
#  -L        to disable the trafficserver logging altogether
```

Example Apache Configuration

```
LoadModule ironbee_module /usr/local/ironbee/lib/mod_ironbee.so
```

```
# IfModule serves for distributions using certain kinds of
# configuration tool.  It's not useful for individual configurations,
# and may make troubleshooting harder.

<IfModule ironbee_module>
    IronbeeConfigFile /usr/local/ironbee/etc/ironbee-httpd.conf

    #Further directives available but having defaults (shown here)
    #likely to work for most users
    IronbeeRawHeaders On
    IronbeeFilterInput On
    IronbeeFilterOutput On
    IronbeeLog On
    IronbeeLogLevel 4
</IfModule>
```

Example IronBee Configuration

```
### Logging
# Log level (standard syslog levels with additional debug2 and debug3)
LogLevel info
```

```
# The log is really only valid for clipp as the servers
# will utilize their own native logging facilities.
Log debug.log

### Sensor Info
SensorId 80ECD8CF-318C-4915-A8C2-B3AAE315FB0C

### Load Modules
# Standard support modules
LoadModule "ibmod_htp.so"
LoadModule "ibmod_pcre.so"

# ### Lua Modules
# # Lua Support
# LoadModule "ibmod_lua.so"
#
# # Event Processor
# LuaLoadModule "event_processor.lua"
# EventProcessorCategoryFilter FOO 60
# EventProcessorCategoryFilter BAR 50
# EventProcessorCategoryFilter BOO 75
# EventProcessorCategoryFilter EEK 75
#
# # Threat Level
# LuaLoadModule "threat_level.lua"
# ThreatLevelMinConfidence 25

# IronBee Rule Language
LoadModule "ibmod_rules.so"

### Auditing
AuditEngine RelevantOnly
AuditLogIndex None
AuditLogBaseDir /tmp/ironbee
AuditLogSubDirFormat "%Y%m%d-%H%M"
AuditLogDirMode 0755
AuditLogFileMode 0644
AuditLogParts all

### Buffering
RequestBuffering On
ResponseBuffering On

### Rule Diagnostics Logging
RuleEngineLogData all
RuleEngineLogLevel info


# ============================================================================
# Rules
# ============================================================================
```

```
# ============================================================================
### Test Rules
# This rule will block if a "blockme" parameter (with any value) is in the request
Rule ARGS:blockme.count() @ne 0 id:test/blockme rev:1 phase:REQUEST "msg:Test blocking" tag:TestRule

### Default Blocking Rules
# These rule detect any advisory blocks and perform the
# actual block.
Rule FLAGS:block @ne 0 id:block/request_header rev:1 phase:REQUEST_HEADER "msg:Blocking request head
Rule FLAGS:block @ne 0 id:block/request rev:1 phase:REQUEST "msg:Blocking request" tag:BlockingMode
Rule FLAGS:block @ne 0 id:block/response_header rev:1 phase:RESPONSE_HEADER "msg:Blocking response h
# ============================================================================
# ============================================================================

### Sites
# Default
<Site default>
    SiteId 0CA1665C-F27F-4763-A3E0-A31A00477497
    Service *:*
    Hostname *

    # Enable rules from the main context
    RuleEnable tag:TestRules
    RuleEnable tag:BlockingMode
</Site>
```

Appendix B: Ideas For Future Improvements

This document contains the list of things we want to look into.

Reminder (to ourselves): We will not add features unless we can demonstrate clear need.

Directive: LoadModule

Support many instances of the same module:

```
LoadModule /path/to/module.so moduleName
```

Module name is optional. When not provided, the filename with extension removed is used as the name.

Some ideas to support module parameters, should we need to do it later on:

```
<LoadModule /path/to/module.so>
    Param paramName1 paramValue1
    Param paramName2 paramValue2

    <Param paramName3>
        # value3 here, free-form
    </Param>

    Param paramName4 @file:/path/to/file/with/paramValue4
</LoadModule>
```

Modules should be able to hook into the engine in the correct order relative to other modules, but should manual tweaking be needed, we could use the following:

```
<LoadModule /path/to/module.so>
    HookPriority hookName PRIORITY_FIRST "beforeModule1, beforeModule2" "afterModule1, afterModule2"
</LoadModule>
```

Directive: RequestParamsExtra

Extract parameters transported in the request URI. The parameter supplied to this directive should be a regular expression with named captures. On a match, the named captures will be placed in the ARGS_EXTRA collection. A new effective path will be constructed (using back references?).

Variable: ARGS_EXTRA

Request parameters extracted from RequestParamsExtra.

transformation: sqlDecode

Decodes input in a way similar to how a DBMS would:

- Decodes hexadecimal literals that use the SQL standard format `x'HEX_ENCODED_VALUE'` (case insensitive)
- Decodes hexadecimal literals that use the ODBC format `0xHEX_ENCODED_VALUE` (case insensitive)
- Decodes backslash-escaped characters

References:

- MySQL Reference: Hexadecimal Literals http://dev.mysql.com/doc/refman/5.6/en/hexadecimal-literals.html
- String Literals http://dev.mysql.com/doc/refman/5.6/en/string-literals.html

Appendix C: Comparison with ModSecurity

IronBee was designed and implemented by the team originally responsible for ModSecurity. Because of this, you will see some similarity in concepts and terms. However, IronBee is not a fork of ModSecurity, but rather a completely new design built on the experience from working on ModSecurity. This section describes both similarities and differences in design, implementation and configuration. Those moving from ModSecurity to IronBee should find this a useful guide to quickly get you started using IronBee.

Design and Implementation

In concept, ModSecurity and IronBee are not that different. Both provide a means for inspecting HTTP transactions. They do this by exposing parsed HTTP data which can be inspected by various means. Actions can then be taken such as logging or interrupting a transaction. In design, however, ModSecurity and IronBee differ greatly.

ModSecurity was designed to be run as an Apache Webserver module and thus requires the Apache Webserver API to execute. While the project has since abstracted this Apache Webserver API and used this to port to other webservers, these ports are still experimental and not full featured. ModSecurity is limited to being configured by the native webserver configuration language.

IronBee was designed as a highly portable framework which can easily be extended and embedded. At the core, IronBee is just a shared library which exposes an API and allows loading external modules to extend functionality. The framework separates data acquisition and configuration from its core. What this means, is that IronBee does not dictate how it acquires HTTP data, nor how you define rules - these are done outside of IronBee. IronBee is configured in its own configuration file, not the configuration of the server in which it is embedded. This has a few benefits: configuration is not limited by the native config language, configuration is not intertwined (potentially conflicting) with the webserver, and IronBee is configured the same (can share configuration) across different server types.

Server plugins load the IronBee library and pass it data. IronBee's rule languages are defined by loadable modules. Because of this, IronBee is capable of being embedded into anything that can pass it data and rules can be written in various different forms - including programmatically. Currently IronBee supports being embedded in Apache TrafficServer, Apache Webserver, Nginx, Taobao Tengine (nginx fork) as well as a command line tool that accepts various formats as input such as raw HTTP, pcap files, etc. Porting to other platforms is relatively simple. IronBee rules can currently be written in a simplistic rule (signature) language, which is similar to ModSecurity, configured via the Lua language (though executed in the C engine), or written in pure Lua. IronBee can be extended with modules written in C, C++ or Lua.

Configuration

Unlike ModSecurity, which is configured within the webserver configuration language, IronBee is configured primarily in its own configuration language, with only a bare minimum configuration done within the server. While there is a full configuration language available for IronBee, the configuration is exposed via an API, so the entire configuration can also be done programatically within a module if so desired (limited support is already available via lua modules). There are two primary differences between configuring ModSecurity and IronBee. IronBee defines sites/locations separate from the webserver, and the IronBee rule language differs from ModSecurity. The differences in the Rule language are discussed here.

Included in the IronBee distribution is a script to translate ModSecurity rules to IronBee syntax. While this script cannot translate some complex logic, it may aide in translation. The script reads stdin and outputs to stdout, translating rules keeping comments and indention intact.

```
./ib_convert_modsec_rules.pl <modsec_rules.conf >ironbee_rules.conf
```

Table C.1. Rule Language Comparison

IronBee	ModSecurity	
General Terminology: • *Data Field:* Data associated with a name. • *Operator:* A named operation performed on a field. • *Modifier:* A generic term for a named item modifying a rule such as rule metadata or an action to execute. • *Metadata:* Name and value associated with a rule. • *Action:* A named operation to be execute based on the rule operator result. Actions can be executed on the operator returning either true or false.	General Terminology: • *Target:* Data associated with a name. • *Operator:* A named operation performed on a field. • *Metadata:* Name and value associated with a rule. • *Action:* A named operation to be execute on the rule operator returning true. Can be either disruptive on non-disruptive.	
General Rule Syntax:	General Rule Syntax:	

`Rule FIELDS @operator param Action modifiers`	`SecRule TARGETS operator actions SecAction actions`
Fields are a whitespace separated list of field specifications, operators take a single parameter, and modifiers are a whitespace separated list of name:value pairs. Example:	Targets are a pipe separated list of target specifications, an operator is a '@' prefixed operator name followed by parameters, and actions are a comma separated list of name:value pairs. Example:
`Rule ARGS:a ARGS:b @rx foo|bar "id:1,msg:'Test rule',log,block"`	`SecRule ARGS:a|ARGS:b "@rx foo|bar" "id:1,msg:'Test rule',log,block"`

Field Transformations:
IronBee allows transformations to be applied to each field individually in addition to using transformations specified as a rule modifier.
Count of variables within the ARGS collection:
`ARGS.count()`
Length of all ARGS starting with an "a":
`ARGS:/^a/.length()`
Trim and lowercase all ARGS named x:
`ARGS:x.trim().lowercase()`
	Target Operators:
	ModSecurity only allows for transformations specified for all targets, except for special length and count target operators.
	Count of variables within the ARGS collection:
	`&ARGS`
	Length of all ARGS starting with an "a":
	`#ARGS:/^a/`

Captured data defaults to CAPTURE collection, but the collection name can be specified. In addition, some operators support capturing all matches, not just the first match.	Captured data is limited to the first 9 captures and is stored in the fixed TX collection. Capture the first element of the path into `TX:1` and the full match into `TX:0`

Capture the first element of the path into `CAPTURE:1` and the full match into `CAPTURE:0`	`SecRule REQUEST_PATH "@rx ^/*([^/]*)" "id:1,capture"`	
	`Rule REQUEST_URI_PATH @rx "^/*([^/]*)" id:1 capture`	
The same as above, but capture to `PATH:1` and the full match into `PATH:0`		
	`Rule REQUEST_URI_PATH @rx "^/*([^/]*)" id:1 capture:PATH`	
Capture all path elements into `PATH` (the @dfa operator finds all matches by default)		
	`Rule REQUEST_URI_PATH @dfa "[^/]+" id:1 capture:PATH`	
Capture anything that looks like a credit card number from the response body into the `CC` collection:		
	`StreamInspect RESPONSE_BODY_STREAM @dfa "\d{15,16}" id:1`	`capture:CC`

Rules

- In chained rules, actions (known as disruptive actions in ModSecurity) are specified on the last rule in the chain, not on the first; the rule metadata remains with the first rule in the chain.
- There are no default transformations, which means that it is no longer necessary to begin rules with `t:none`.
- The `SecRuleScript` functionality is now handled by `RuleExt`, which works as an interface between the rule language and externally-implemented security logic (for example, Lua rules, rules implemented natively, etc).
- Run-time rule manipulation (using `ctl:ruleRemoveById` or `ctl:ruleUpdateTagetById`) is not currently supported. These features are slow in ModSecurity and we wish to rethink them before and if we implement them. At the very least we will wish to provide a faster implementation.

- Changing rule flow at run-time is not supported at this time. This means that the functionality of `skip`, `skipAfter` is not supported. This is not a likely feature to be implemented. Instead you are encouraged to use Lua.
- IronBee uses a simplified configuration model in which site configuration always starts from scratch. Inheritance is used when location contexts are created, but, unlike in ModSecurity, locations always inherit configuration from their sites.
- There is no ability to update rule targets and actions at configure-time, but we will probably implement a similar feature in the future.
- All rules that generate events must use the `msg` modifier to provide a message. This is because IronBee does not use machine-generated rule messages. In ModSecurity, machine-generated messages have shown to have little value, especially as rules increase in complexity. They are often a source of confusion.

Miscellaneous

- The audit log format has been redesigned to support new features.
- In IronBee, request and response body inspection is not tied to buffering. Disabling buffering will generally not affect inspection; it will only affect the ability to block attacks reliably.
- In IronBee, like in ModSecurity, you can have a transaction blocked if the buffer limit is encountered, but, you can also choose to continue to use the buffer in circular fashion. In that case, IronBee will simply buffer as much data as it can, allowing any overflowing data to pass through.

Features Not Supported (Yet)

- Content injection - will be added in the future
- Guardian log - will not implement this obsolete feature
- XML support - will be added in the future
- Chroot support - will not implement this `httpd`–specific feature
- File upload inspection and extraction - will be added in the future
- Anti-DoS features are not supported, and we don't expect they will be in the future