OWASP
The Open Web Application Security Project

OWASP ESAPI for Java EE 2.0a

Web Application Firewall Policy File Specification

Foreword

This document provides instructions for configuring the Web Application Firewall (WAF) feature of the Java EE language version of the OWASP Enterprise Security API (ESAPI). The ESAPI WAF provides application developers with the capability to deploy virtual patches for security vulnerabilities until underlying code can be fixed. It is implemented as a Java EE filter, which offers many advantages when compared to many commercial and open source WAF alternatives.

We'd Like to Hear from You

Further development of ESAPI occurs through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. Please address comments and questions concerning the API and this document to the ESAPI mail list, owasp-esapi@lists.owasp.org

Copyright and License

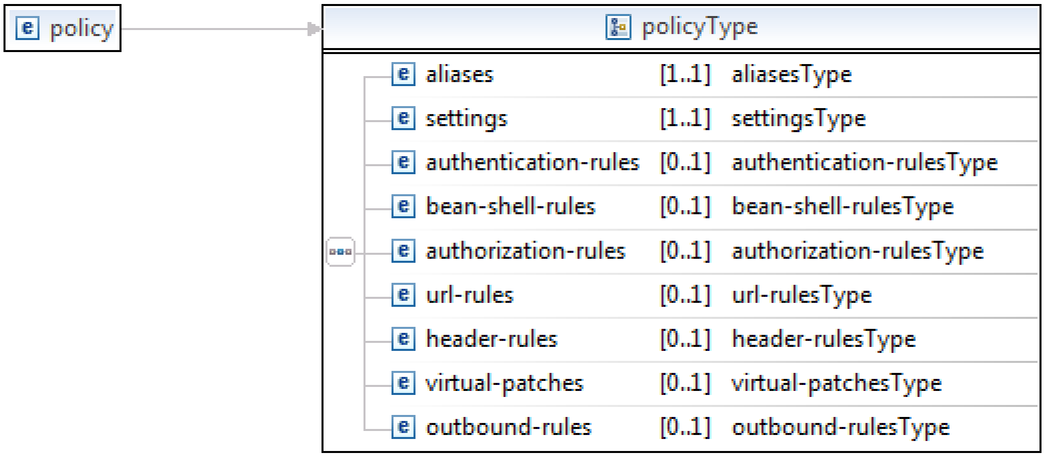
This page is intentionally blank

1. Table of Contents

2. The policy file

The ESAPI Web Application Firewall (WAF) is driven by an XML policy file that tells it what rules to enforce in the application. These rules can do a number of things, from simple virtual patching to complex authorization enforcement with BeanShell scripts.

This document describes the structure of the policy file, the individual rules and how they work. There are also a number of examples in order to guide you during implementation. The following picture shows you a visual representation of the policy file XSD, a formal specification for the layout of a policy file:



aliases	[1..1]	aliasesType
settings	[1..1]	settingsType
authentication-rules	[0..1]	authentication-rulesType
bean-shell-rules	[0..1]	bean-shell-rulesType
authorization-rules	[0..1]	authorization-rulesType
url-rules	[0..1]	url-rulesType
header-rules	[0..1]	header-rulesType
virtual-patches	[0..1]	virtual-patchesType
outbound-rules	[0..1]	outbound-rulesType

As is seen, a policy file’s root element is **policy**, with no attributes. Inside the root **policy** element are a number of **rule sections**. These are just categories of rules to help organize the policy file. As the middle column shows, only the **aliases** and **settings** sections are required. With that in mind, here is an example of a skeleton policy file with no rules to enforce. The meanings of the values in the **settings** section will be described later.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<policy>

   <aliases></aliases>

   <settings>
      <mode>block</mode>
      <session-cookie-name>JSESSIONID</session-cookie-name>
      <error-handling>
         <default-redirect-page>/error.jsp</default-redirect-page>
         <block-status>500</block-status>
      </error-handling>
   </settings>

</policy>
```

If you’re writing your own policy file it may be useful to start off with the “empty” file above and slowly integrate rules. Each rule has an optional **id** attribute that can be used to uniquely identify every rule. While not required, these **id** values can help you identify the rules that are fired in the logs.

The rest of the document is dedicated to discussing each of the **rule sections** shown above.

3. Aliases section

The <aliases> section lets you define common strings for use throughout the policy file. It can have 0 or more <alias> elements within it.

Imagine a number of vulnerabilities have been found in one part of the site, within /admin/. Since there are a number of rules that are going to reference that portion of the site, it's useful to have a single place to define that section of the site. It reduces the opportunity for mistyping a rule and helps make rules more organized.

Here's an example of an <aliases> section that only defines one alias:

```
<aliases>
   <alias name="ADMIN_PATH" type="regex">^/admin/.*</alias>
</aliases>
```

The `name` attribute (required)

The **name** of an alias is the identifier by which the alias will be referenced later on in the policy. It should be something easy for a human to understand.

The `type` attribute (optional)

The **type** attribute tells the WAF if the string is a regular expression of a string literal. If none is specified, the string is assumed to be a string literal. A value of "regex" tells the WAF that the string represents a regular expression. It should be noted that the regular expression is written with Java regular expression syntax.

4. Settings

The <settings> section is where the overall configuration settings of the WAF are set. The WAF basically must learn two things after parsing this section: what **mode** WAF is the in, and how to perform **error-handling**. Here's an example of a <settings> section:

```
<settings>
   <mode>redirect</mode>
   <error-handling>
      <default-redirect-page>/security/error.jsp</default-redirect-page>
      <block-status>403</block-status>
   </error-handling>
</settings>
```

The `mode` value (required)

The **mode** indicates at a high level how to handle serious security events. Security events occur when certain rule types match an incoming request. For instance, if a user trips a **<virtual-patch>** rule then the application will do what the **mode** indicates. However, an **<add-header>** rule, which is intended to be fired on every request, is usually an implementation of a security best practice doesn't require any other action to occur.

There are 3 legal values for this field: **redirect**, **block**, and **log**.

The `log` mode (default)

When mode is set to **log**, or any other value, the WAF only performs logging and does not perform any other action when a security rule is tripped. This mode is useful for testing rules in production in order to calibrate rules against false positives during a short "trial" period.

Setting the mode to any other value will not prevent the WAF from logging in the exact same way as would occur if the mode were to set **log**. Specifying this only tells the WAF not to perform any other actions.

The `redirect` mode

When the mode is set to **redirect** the application will redirect users to an error page with a 302 or JavaScript client-side redirect, depending on the state of the application when the rule is tripped. The URL to which the user is sent is set in the **error-handling** section.

The `block` mode

When the mode is set to **block** the application will simply stop processing the request and return a blank response if a serious security error has occurred.

The `session-cookie-name` value (optional)

The **session-cookie-name** value tells the WAF what the application container's session cookie name is. If none is specified it is assumed to be JSESSIONID. This field is only used for any experimental functionality and may be required in the future.

The `error-handling` values

There are two values within the **error-handling** block within **settings**: the **default-redirect-page** value and the **block-status** value.

The `default-redirect-page` value

This URL is where users are redirected after a serious security event occurs when the WAF is in **redirect** mode. It can be a relative or fully-qualified URL and will be used in a 302 or JavaScript on the client side. It is recommended that this value be set to the location for a generic error page.

The `block-status` value

This value is an integer which indicates the HTTP code to be used when a serious security event occurs while the WAF is in **block** mode. Typical values may be 401, 403 or 500. This value has no effect when the WAF is in **log** or **redirect** mode.

5. Authentication rules

The <authentication-rules> section allows the WAF to enforce typical J2EE authentication requirements.

A typical J2EE application authentication pattern involves making sure a session variable exists inside some base action class. If the variable doesn't exist, the request is considered to be unauthenticated and is handled appropriately. However, because applications support such a wide range of functionality and partner services, mistakes are often made in protecting all functionality that requires authentication.

If this pattern can't be used to authenticate users in your application, then you should not use this rule type. Here is an example of an <authentication-rules> section that protects all of an application's URL endpoints except for a few things that are meant to be public:

```
<authentication-rules path="/.*" key="UserAuthKey" >
   <path-exception>/</path-exception>
   <path-exception>/index.html</path-exception>
   <path-exception>/login.jsp</path-exception>
   <path-exception>/index.jsp</path-exception>
   <path-exception type="regex">/images/.*</path-exception>
   <path-exception type="regex">/css/.*</path-exception>
   <path-exception type="regex">/help/.*</path-exception>
</authentication-rules>
```

The `path` attribute (required)

The **path** attribute tells the WAF what portion of the site requires authentication, in the form of a regular expression. In the preceding example, the value of **path** is "/.*", which means that authentication will be forced on any request that reaches the application.

The `key` attribute (required)

The **key** attribute tells the WAF what session attribute should be checked for existence. If this session attribute has any non-null value, the user will be assumed to be authenticated.

Path exceptions (optional)

There are a number of rules that can have nested **path-exception** values. A **patch-exception** tells the WAF not to apply the given rule to a particular path. If the "type" attribute of the **path-exception** is set to "regex", the path is assumed to be a regular expression, otherwise it is assumed to be a string literal.

These **path-exception** values are used to unprotect static content, marketing material, and anything else a user should be able to access without logging in.

6. Authorization rules

The <authorization-rules> section allows the WAF to enforce typical J2EE authorization requirements. This section supports two different types of rules: **restrict-source-ip** rules and **must-match** rules.

The `restrict-source-ip` rule

Most applications are have a “normal” user interface and an “admin” user interface. Since they are deployed together in the same application, “normal” users can access the administrator portion if they can guess the correct URLs or steal the credentials of an administrator.

The **restrict-source-ip** rule helps with that problem by restricting access to certain paths in the application according to IP. Enforcing this segmentation helps limit the damage done by exposed administrator credentials and allows fine-grained network policy for reaching different tiers of an application.

Here is an example **restrict-source-ip** rule:

```
<authorization-rules>
   <restrict-source-ip
      type="regex"
      ip-header="X-ORIGINAL-IP"
      ip-regex="(192\.168\.1\..*|127.0.0.1)">/admin/.*</restrict-source-ip>
</authorization-rules>
```

In this example our <authorization-rules> element has one entry, a **restrict-source-ip** rule that restricts access to any URL beginning with /admin/ to those users coming from an IP that is either inside the local LAN (192.168.1.*) or from the server itself.

The `type` attribute (optional)

The **type** attribute, when set to “regex”, tells the WAF that the path to be protected is a regular expression as opposed to a string literal.

The `ip-header` attribute (optional)

The **ip-header** attribute tells the WAF the request header that will hold the original user’s true IP address. Applications are often setup downstream of one or multiple proxy servers that would mask the true original IP of the request. In this case, appliances often populate a request header with the true original IP address of the request. In the preceding example, the user’s IP address can be found in the *X-ORIGINAL-IP* header.

Not specifying this will tell the WAF to use the IP address found in the **HttpServletRequest** object.

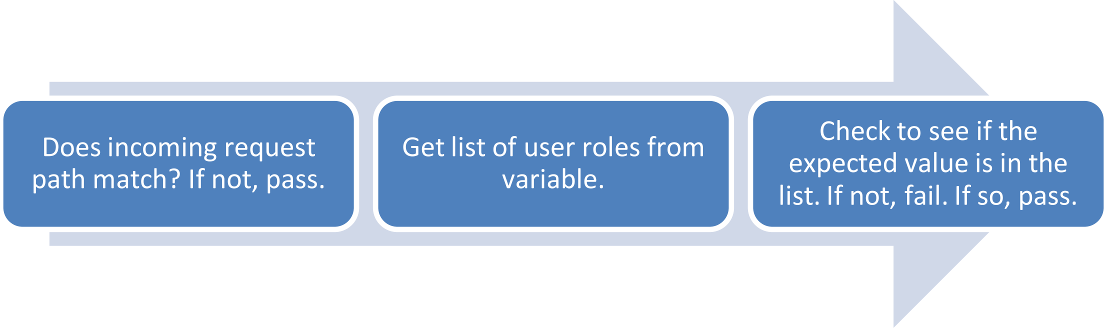
The `ip-regex` attribute (required)

The **ip-regex** attribute should contain a regular expression that, when matched to an incoming request IP address, should indicate that access be granted. In simple terms, you want to match the people *you want to access* the sensitive part of the application, not the opposite.

The `must-match` rule

The **must-match** rule is a powerful construct that can be used to implement a number of different functions, most notably authorization; specifically role-based access control checks.

Technically, the **must-match** rule works like this:



Does incoming request path match? If not, pass.

Get list of user roles from variable.

Check to see if the expected value is in the list. If not, fail. If so, pass.

There are generally two places a J2EE application will look for a user's roles when performing authorization checks; in the session or in a request header. In the following example, the WAF will check to see if a request header, called, *X-ROLES* (case sensitive), contains a substring "admin". This pattern is often used when an appliance (most notably CA SiteMinder) supplies trusted header values to applications behind its perimeter.

```
<authorization-rules>
   <must-match
      path="^/admin/.*"
      variable="request.headers.X-ROLES"
      operator="contains"
      value="admin" />
</authorization-rules>
```

Here is an example of the alternative, where user roles are stored in a session variable called *org.acme.user.roles*:

```
<authorization-rules>
   <must-match
      path="^/admin/.*"
      variable="session.org.acme.user.roles"
      operator="inList"
      value="admin" />
</authorization-rules>
```

In this example, the **operator** has been changed to *inList*, which will inspect the session variable set in **variable**. The WAF knows how to inspect any subclass of **java.util.Collection**, **java.util.Enumeration** and

java.util.Map, which include all the basic List objects. So, if you store your user roles in any of those subclasses, the WAF can search through it for the value specified by the **value** attribute.

The `path` attribute (required)

The **path** attribute is a regular expression that decides which paths in the application to which this **must-match** rule will apply.

The `variable` attribute (required)

The **variable** attribute tells the WAF where to search for the **value** attribute in order to grant access. There are 5 places it understands where to find data:

Prefix	Description
request.parameters.some_parameter	A request parameter, whether or not it came from the URL, POST or multipart data.
request.headers.some_header	A request header.
request.uri	The request URI, not including any querystring data.
request.url	The request URL, including any querystring data.

The `operator` attribute (required)

The **operator** attribute tells the WAF what kind of operation to perform on the **value** attribute in respect to the **variable** attribute. There are 4 legal values:

Operator	Description
equals (default)	A simple String equals test between the **variable** and **value**.
exists	Checks to see if the **variable** is not null.
inList	Checks to see if the **value** is equal to an entry in the **variable** list-type object.
Contains	Checks to see if the **value** is a substring of the **variable**.

The value attribute (required)

The **value** attribute indicates the *value that you're looking for* in order to pass the test.

7. URL rules

The <url-rules> section allows a developer to perform checks on data that is sent in the HTTP status line, the first line of the request. In this section you can restrict access based on file extension requested (**restrict-extension**), the HTTP method requested (**restrict-method**) and whether or not the request is over SSL (**enforce-https**).

The `restrict-extension` rule

The **restrict-extension** allows a developer to restrict access to certain file extensions on the server. Assessments often find that sensitive static resources like code or libraries are found on the server unprotected. If those problems are systemic or it is challenging to physically remove the files from the server, you can use the **restrict-extension** rule.

The following example shows how to prevent any request for Java source files that were accidentally left on the production server:

```
<url-rules>
  <restrict-extension deny=".java" />
</url-rules>
```

The `deny` and `allow` attributes (one is required)

A **restrict-extension** rule can have either **deny** or **allow** attributes, but not both. It is possible to have multiple **restrict-extension** rules in place to have that intended effect, however. The value put into the attribute chosen will be used to build a regular expression of the form: “.*\<value here>$”.

There are pros and cons of whitelisting (using the **allow** attribute) and blacklisting (using the **deny** attribute) attribute. Depending on the circumstances of the problem, either answer may be preferable.

The `restrict-method` rule

By default, most application servers allow a number of methods that can be abused by users to allow users to cause unintended actions. The **restrict-method** rule allows a developer to prevent requests with unwanted methods from reaching a portion of the application.

The following examples shows a <url-rules> section with two **restrict-method** rules:

```
<url-rules>
   <restrict-method deny="GET" path=".*\.do$" />
   <restrict-method allow="^(GET|POST|HEAD)$" />
</url-rules>
```

The `deny` and `allow` attributes (one is required)

A **restrict-method** rule can have either **deny** or **allow** attributes, but not both. It is possible to have multiple **restrict-method** rules in place to have that intended effect, however. The value put into the attribute chosen is a regular expression test.

There are pros and cons of whitelisting (using the **allow** attribute) and blacklisting (using the **deny** attribute) attribute. Depending on the circumstances of the problem, either answer may be preferable.

The `path` attribute (optional)

The **path** attribute gives the WAF a regular expression to decide when to apply this rule. If the URL matches the **path** regular expression, the rule will be tested.

The `enforce-https` rule

Applications should always try to use SSL, whenever possible. If the application uses relative links and doesn't enforce the use of SSL, it's possible than an attacker could trick a victim into clicking on a non-SSL link, possibly causing them to expose sensitive information.

To prevent this, there is an **enforce-https** rule. The following example shows how to enforce the use of SSL throughout the site, with a few simple exceptions for static, non-sensitive content. If the user makes a non-SSL request for a path that matches the rule, they will be redirected to the same URL, but over SSL, in a client-side 302 redirect.

```
<url-rules>
   <enforce-https path="/.*">
     <path-exception>/index.html</path-exception>
     <path-exception type="regex">/images/.*</path-exception>
     <path-exception type="regex">/help/.*</path-exception>
   </enforce-https>
</url-rules>
```

The `path` attribute (required)

The **path** attribute dictates what portion of the site should enforce the usage of SSL. Any exceptions should be listed in the **path-exception** values nested inside of the **enforce-https** element.

Path exceptions (optional)

There are a number of rules that can have nested **path-exception** values. A **patch-exception** tells the WAF not to apply the given rule to a particular path. If the "type" attribute of the **path-exception** is set to "regex", the path is assumed to be a regular expression, otherwise it is assumed to be a string literal.

These **path-exception** values are used to unprotect static content, marketing material, and anything else a user should be able to access over an insecure channel.

8. Header rules

The <header-rules> section allows a developer to perform checks on data that is sent in HTTP request headers. In this section you can restrict access based on user agent (**restrict-user-agent**) or content-type (**restrict-content-type**).

The `restrict-user-agent` rule

The **restrict-user-agent** allows a developer to restrict access to the site according to user agent. There are some user agents that indicate openly unwanted traffic, like automated search engine robots. Also, novice attackers will sometimes use attack tools that broadcast their nature in the user agent header.

The following example shows two **restrict-user-agent** rules that prevent Google robot traffic and allow all other traffic:

```
<header-rules>
   <restrict-user-agent deny=".*GoogleBot.*" />
   <restrict-user-agent allow=".*" />
</header-rules>
```

The `deny` and `allow` attributes (one is required)

A **restrict-user-agent** rule can have either **deny** or **allow** attributes, but not both. It is possible to have multiple **restrict-user-agent** rules in place to have that intended effect, however. The value put into the attribute chosen is a regular expression.

There are pros and cons of whitelisting (using the **allow** attribute) and blacklisting (using the **deny** attribute) attribute. Depending on the circumstances of the problem, either answer may be preferable.

The `restrict-content-type` rule

The **restrict-content-type** allows a developer to restrict access to the site according to the request content type. Abnormal content-types can be indicators of CSRF attacks and can be used to try to bypass security mechanisms.

The following example shows two **restrict-content-type** rules that prevent multipart traffic and allow all other text traffic:

```
<header-rules>
   <restrict-content-type deny=".*multipart.*" />
   <restrict-content-type allow=".*" />
</header-rules>
```

The `deny` and `allow` attributes (one is required)

A **restrict-content-type** rule can have either **deny** or **allow** attributes, but not both. It is possible to have multiple **restrict-content-type** rules in place to have that intended effect, however. The value put into the attribute chosen is a regular expression.

There are pros and cons of whitelisting (using the **allow** attribute) and blacklisting (using the **deny** attribute) attribute. Depending on the circumstances of the problem, either answer may be preferable.

9. Virtual patches

The <virtual-patches> section allows a developer to implement virtual patches against known attacks, like unchecked redirects, XSS and SQL injection, among others. It can have 0 or more **virtual-patch** rules.

The `virtual-patch` rule

The **virtual-patch** rule is one of the most important rules in the WAF policy. Many attacks discoverable with automated tools can be fixed with a **virtual-patch**. The following example of a virtual patch patches a simple cross-site scripting (XSS) vulnerability:

```
<virtual-patch
   id="scr-15520"
   path="/foobar.jsp"
   variable="request.parameters.q"
   pattern="^[0-9a-zA-Z\s,\.]$"
   message="detected exploit of SCR #15520" />
```

The `id` attribute

The **id** attribute should uniquely identify a particular **virtual-patch**. If a unique ID is used, log message will be differentiable according to which rules were tripped.

The `path` attribute

The **path** attribute is a regular expression applied to the request URI. If it matches the rule will be tested.

The `variable` attribute

The **variable** attribute tells the WAF where to analyze with the **pattern** attribute. There are 2 places it understands where to find data:

Prefix	Description
request.parameters.some_parameter	A request parameter, whether or not it came from the URL, POST or multipart data.
request.headers.some_header	A request header.

The `pattern` attribute

The **pattern** attribute is a regular expression applied to the value of the **variable** attribute. If the **pattern** matches successfully, the request passes the test. If the **pattern** doesn't match the requested value, it is considered to be a failure, and the WAF will recognize it as a serious security event.

The `message` attribute

The **message** attribute is a literal string to be logged when the virtual-patch fails (indicating an attack). It should be noted that the logger will automatically generate some of the most useful information automatically, and this should be used to indicate to a human a succinct summation of what this rule failure indicates.

10. Outbound rules

There are a number of things an application security expert will recommend that will require access to outbound data before it is delivered to the end user. Some of these things include adding additional headers to prevent caching or cross-domain attacks (**add-header)**, or egress filtering to prevent data loss (**detect-content** and **dynamic-insertion**). They may also include adding flags to container cookies (**add-http-only-flag** and **add-secure-flag**). All of these are handled within the <outbound-rules> section.

The `add-header` rule

The **add-header** rule allows a developer to add custom headers to responses according to the path requested.

The following example shows an **add-header** rule that tells IE8 browsers that the response should not be allowed to be framed by any domain other than the current one:

```
<add-header name="X-FRAME-OPTIONS" value="SAMEORIGIN" path="/.*">
   <path-exception type="regex">/frame_me/.*</path-exception>
</add-header>
```

The `path` attribute

The **path** attribute is a regular expression applied to the request URI. If it matches the header will be added, provided the **path-exception** values don't match the **path**.

The `name` attribute (required)

The **name** attribute contains the name of the header to be added.

The `value` attribute (required)

The **value** attribute contains the value of the header to be added.

Path exceptions (optional)

There are a number of rules that can have nested **path-exception** values. A **patch-exception** tells the WAF not to apply the given rule to a particular path. If the "type" attribute of the **path-exception** is set to "regex", the path is assumed to be a regular expression, otherwise it is assumed to be a string literal.

These **path-exception** values are used to unprotect public content.

The `detect-content` rule

The **detect-content** rule allows a developer to detect when certain data or patterns of data are sent to the browser. Here is an example of a **detect-content** rule used to see when old dates are being sent to the users in legally binding disclaimers:

```
<detect-content content-type=".*text/.*" pattern=".*2008.*" />
```

In more painfully realistic situations, developers may be interested in patterns being sent to the browser that indicate a loss of sensitive data, like SSNs, credit cards, or other custom classified information.

The `content-type` attribute (required)

The **content-type** attribute allows the WAF to apply the **detect-content** rule only when the content type of the outbound response matches the **content-type** attribute. This value is a regular expression. For example, using a **content-type** of ".*text/.*" will make sure performance of the WAF won't suffer as the rule searches through binary data.

The `path` attribute (optional)

The **path** attribute is a regular expression applied to the request URI. If it matches the rule will be tested.

The `pattern` attribute (required)

The **pattern** attribute holds the regular expression that will detect the data the developer is looking to discover.

The `dynamic-insertion` rule

The **dynamic-insertion** rule allows a developer to dynamically modify outbound response body data sent to the browser. This can be used to in incident response or preventing information leakage.Here is an example of a **dynamic-insertion** rule used to prevent commented stack traces from being sent to the user:

```
<dynamic-insertion
   path="/err.jsp"
   pattern="&lt;!-- BEGIN STACK TRACE.*--&gt;">
      <replacement><![CDATA[stack trace removed]]></replacement>
</dynamic-insertion>
```

The `content-type` attribute (required)

The **content-type** attribute allows the WAF to apply the **dynamic-insertion** rule only when the content type of the outbound response matches the **content-type** attribute. This value is a regular expression. For example, using a **content-type** of ".*text/.*" will make sure performance of the WAF won't suffer as the rule searches through binary data.

It should be noted that if no content-type has been set on a response before it gets to the WAF, the WAF will assume it is ISO-8895-1.

The `path` attribute (optional)

The **path** attribute is a regular expression applied to the request URI. If it matches the rule will be tested.

The `pattern` attribute (required)

The **pattern** attribute holds the regular expression that will detect the data the developer is looking to replace.

The `replacement` value (required)

The **replacement** value holds the data the developer is looking to replace the detected content with. One can use replacement groups within the detection pattern and replacement string, but this could allow abuse if not done correctly.

A note about `dynamic-insertion` and `detect-content` rules

These rules only work if applications don't try to manage their own output streaming. For instance, if an application calls flush() on its response object to partially push results to the user, the WAF will not have access to the entire response when the "checking" of these two types of rules occur. The rules will claim the sought after content is not present since there is nothing in the buffer to check.

The `add-http-only-flag` rule

The **add-http-only-flag** rule allows a developer to automatically add the HTTPOnly flag to custom cookies.

The following example shows an **add-http-only-flag** rule that tells the WAF to make sure the HTTPOnly flag is applied to all cookies:

```
<add-http-only-flag>
   <cookie name=".*" />
</add-http-only-flag>
```

The `cookie` value (required)

As is shown, the **add-http-only-flag rule** has no attributes, except for the always optional **id** attribute. It does, however, have zero or more cookie elements nested within it. Those **cookie** elements themselves only have one attribute, **name**.

The `name` attribute (required)

The **name** attribute is a regular expression that should match 1 or more of the cookies that require the HTTPOnly flag. Because you can have multiple **cookie** elements, one does not need to create a complex regular expression to match them all – each cookie can have its own element and it can be written as a string literal. However, it was made a regular expression in order to make adding all cookies, regardless of their runtime name, have the flag added.

The `add-secure-flag` rule

The **add-secure-flag** rule allows a developer to automatically add the secure flag to custom cookies.

The following example shows an **add-secure-flag** rule that tells the WAF to make sure the secure flag is applied to all cookies:

```
<add-secure-flag>
   <cookie name=".*" />
</add-secure-flag>
```

The `cookie` value (required)

As is shown, the **add-secure-flag rule** has no attributes, except for the always optional **id** attribute. It does, however, have zero or more cookie elements nested within it. Those **cookie** elements themselves only have one attribute, **name**.

The `name` attribute (required)

The **name** attribute is a regular expression that should match 1 or more of the cookies that require the secure flag. Because you can have multiple **cookie** elements, one does not need to create a complex regular expression to match them all – each cookie can have its own element and it can be written as a string literal. However, it was made a regular expression in order to make adding all cookies, regardless of their runtime name, have the flag added.

A note about adding flags to session cookies

While adding flags is fairly easy for the WAF to accomplish with custom cookies added by standard J2EE methods, it is fairly difficult to apply to container cookies, like an application server's standard session cookie.

Because the standard session cookie is not created during the lifetime of the WAF instance or within its technical capability to alter, the WAF must go to considerable lengths to make this happen. Given different browser behavior regarding cookie management, this issue scales up in complexity quickly. A WAF solution would require multiple (up to 6) separate HTTP communications in order to reliably attach HTTPOnly or the secure flag to session cookies.

Rather than attempt to manage this complexity, it was decided that the WAF should only add these flags to cookies it produced with standard J2EE methods. Most application containers allow developers to add this flag to their session cookies through configuration.

11. Bean shell rules

The <bean-shell-rules> section allows a developer to specify zero or more BeanShell scripts to be run for complex attack detection. Some security decisions aren't simple enough to be made from an XML-driven configuration file, no matter how flexible. The **bean-shell-script** rule allows developers to truly implement code at runtime to perform complex defense.

The following is an example of a **bean-shell-rules** section with one **bean-shell-script** rule defined:

```
<bean-shell-rules>
   <bean-shell-script
      id="example1"
      file="src/WAF/bean-shell-rule.bsh"
      stage="before-request-body"/>
</bean-shell-rules>
```

The `bean-shell-script` rule

The **bean-shell-script** rule is a powerful rule for scripting defensive code in Java without any code changes.

The `id` attribute (required)

The **id** attribute, although usually optional, is intended to be used as a unique identifier for BeanShell scripts and is thus required.

The `file` attribute (required)

The **file** attribute should be a relative path from the current working directory that contains the BeanShell script to be executed.

The `stage` attribute (required)

The **stage** attribute indicates during what stage the BeanShell script located at the **file** attribute should be executed. There are 3 legal values for this field:

Stage value	Description
before-request-body	The script will be executed before any multipart POST data is read, but all normal request.getParameter() values will be accessible.
after-request-body	The script will be executed after any multipart POST data is read. All normal request.getParameter() values as well as multipart form fields will be available.
before-response	The script will be executed directly before the response is sent to the user.

What does a BeanShell script look like?

Understanding BeanShell itself is outside of the scope of this document, but a BeanShell script is essentially just free-standing Java code. Your BeanShell scripts will need access to some contextual data in order to perform security checks, so there are a number of objects exposed to each BeanShell script:

Variable name	Class	Description
request	The request for which this rule is firing.	javax.servlet.http.HttpServletRequest
response	The response associated with this request. If the user has WAF rules that require access to outbound data, the ESAPI WAF response class will be used. Otherwise, the normal J2EE instance is in this placeholder. If you're not sure which you'll be using and you need access to a particular one, you can use the *instanceof* Java keyword to figure out which one you have. In the script.	javax.servlet.http.HttpServletRequest or org.owasp.esapi.waf.internal. InterceptingHTTPServletResponse
session	The session object associated with this request. If this is the first request in the session, this value may be null.	javax.servlet.http.HttpSession
action	This variable, when passed into the script, is initially null. There is a contract in place regarding its usage. If you assign a valid Action subclass to this variable, that action will be performed at the end of the rule. This allows developers to execute customized actions based on the result of the BeanShell script. If this variable is still null after the BeanShell script executes, no action will be taken. For more information on the Action subclasses, see the JavaDoc for ESAPI (specifically org.owasp.esapi.actions.*).	org.owasp.esapi.waf.actions.Action

A WAF BeanShell script should consist of 3 basic sections:

1. **Imports**. Import all the classes to perform your security check.
2. **Do what you want**. Perform the check you want
3. **Tell the WAF the result**. By setting the **action** variable to a new Action subclass, you tell the WAF what you want to do.