Active Content Filter (ACF)

Tokyo Research Laboratory

1. Overview

Cross Site Scripting (XSS) is an attack using a vulnerability that exists implicitly in HTML document processing in Web browsers at the client side. The vulnerability is basically cased by injecting malicious active content such as such as JavaScript, Applets, or ActiveX objects. Various types of XSS attacks are summarized in the Cross Site Scripting website.

Adobe Flash (including Shockwave Flash and Macromedia Flash), JSON, and RSS/ATOM feeds have also become tools for XSS attacks recently, since they are used in Web applications. Flash includes several multimedia Internet technologies that are widely used to add animation and interactive user interfaces to webpages. A Flash object (usually stored in a file with the extension .swf, and called an SWF file) can be directly loaded by a Web browser or integrated into an HTML document. An SWF file may consist of various multimedia objects (movies, pictures, or music) and bytecode for programs written in ActionScript, a scripting language based on ECMAScript. By using its API, the ActionScript in an SWF file can interact with HTML pages or Web servers on a network. Malicious Web users can thus inject malicious content via Flash. See the Cross Site Flashing website.

Some Web 2.0 applications embed JSON values passed from the clients without any HTML encoding into their response HTML document. Agile software development using Ajax and PHP is becoming more popular and is vulnerable to this approach. This may allow XSS attacks by malicious Web users. See the JSON website for examples.

Some Web 2.0 client applications uses RSS/ATOM feeds to exchange data with servers. If the RSS/ATOM feeds include some HTML fragments, they should be HTML escaped property. If not, it would become a new XSS vulnerability.

In order to prevent such XSS attacks, our Active Content Filter (ACF) provides a function to strip out the active content from input. ACF can be either embedded in an application server or placed in a separate proxy server, as shown in Figure 1.

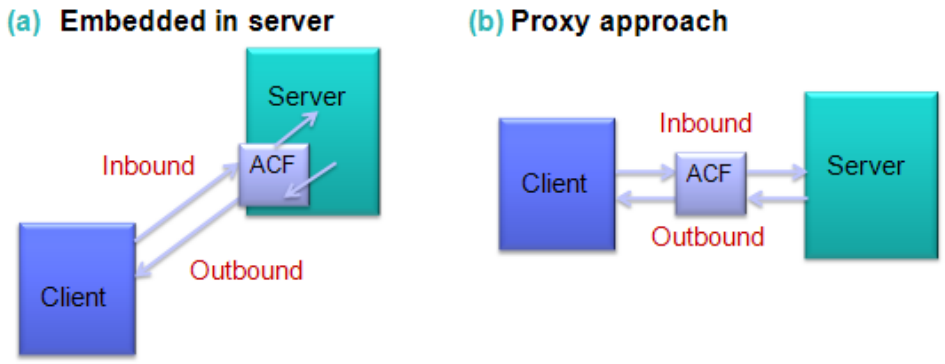

Figure 1: ACF usage scenarios

For HTML documents, ACF is based on an API for streamlined XML/HTML processing, the SAX API. As the SAX-based parser reads the input document, it emits a sequence of events, each of which corresponds to a fragment in the document (such as a start tag or text node). The filter part in ACF receives this sequence and discards the events corresponding to elements containing active content. The other events are passed to the serializer, which serializes the events back as a byte array (see Figure 2).

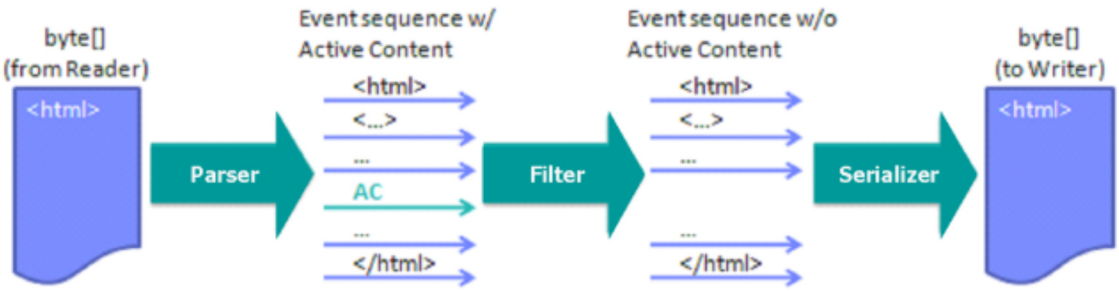

Figure 2: SAX-based HTML processing

2. Supported functions

The current ACF provides the following functions.

1. ***HTML filtering:*** Removes active content from an HTML document or an HTML fragment in an HTTP request parameter. The scope of the filtering can be limited with XPath or by annotation in the HTML document. ACF also supports Flash filtering as an additional function of HTML filtering.
2. ***JSON filtering:*** Removes active content from a JSON value or a JSON-formatted string.

3. Architecture

3.1 Architecture of HTML filtering

When the ACF filters HTML documents, it uses a SAX-based parser to parse the documents. Because the parser provides a filter chain, it allows using custom filters in the chain. To use a custom filter, the custom filter simply receives the incoming SAX-like event sequence and emits the filtered SAX event sequence. During the process, the filter can add new SAX events into the sequence, remove SAX events from the sequence, or modify SAX events in the sequence (see Figure 3). Filters are statically configured using an XML file, as explained in the configuration file for HTML filtering section.

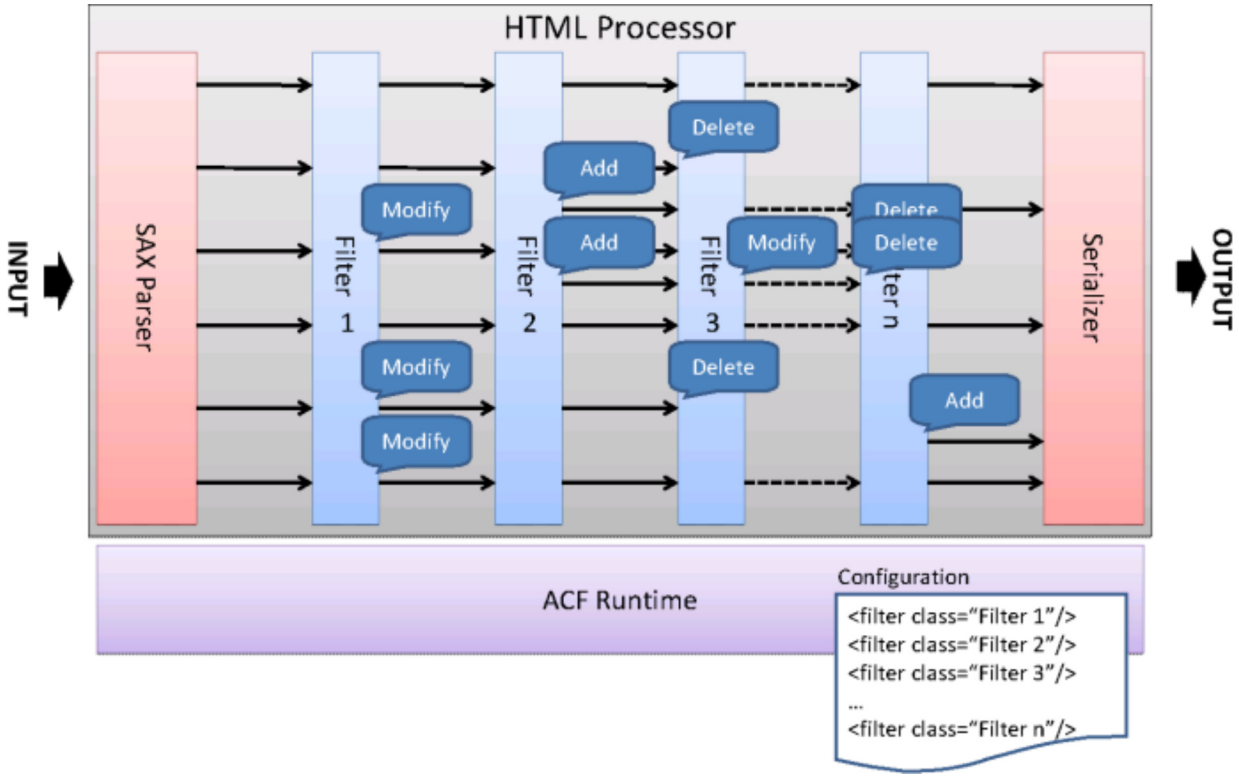

Figure 3: Architecture of HTML filtering

The ACF provides the following default filters.

Attribute value canonicalizer (`com.ibm.trl.acf.impl.html.canonicalizer.C14NFilter`)

This filter works on the `startElement` events. It checks all of the attributes in turn and, if necessary, it decodes their values that have been encoded in various ways such as Base64 or by using entity references. Attackers often try to inject scripts using complex encodings that browsers can interpret, but which are ignored by standard content filters. The attribute value canonicalizer is intended to block such attempts. It should be invoked before the HTML base filter is called. However, once it is invoked, the original information about the input HTML document is lost. For example, any upper case letters used in element names and attribute names are converted to their lower case letter respectively. Various encoding techniques used in the attribute values are decoded.

HTML base filter (`com.ibm.trl.acf.impl.html.basefilter.BaseFilter`)

This filter works on the HTML fragments that include active content. The HTML base filter supports both XPath-based filtering and annotation-based filtering. But **since XPath-based filtering is a legacy support, we recommend using annotation-based filtering.** When annotation-based HTML filtering is used, it does not begin active filtering until it finds a start annotation that specifies a start point for the ACF processing. Once it finds the start point annotation, it continues to filter the current node until it finds the end annotation that specifies the end point of the ACF processing. While filtering, it can remove the current node or remove a malicious attribute value in the node. It is possible to configure the HTML base filter to preserve the original information about the input HTML document. For these configuration settings, please see the configuration file for HTML filtering section.

Flash filter (`com.ibm.trl.acf.impl.html.flashfilter.FlashFilter`)

This filter works on the HTML fragments that contain embedded Flash content. It first determines the filtering scope as defined by XPath or annotation. For the nodes in that scope, it can rewrite each node or remove malicious request parameters passed to the Flash. It is possible to configure this filter to preserve the original information about the input HTML document. For these configuration settings, please see the configuration file for HTML filtering section.

There will be the following filters although the current ACF does not support them.

Word sanitizer

This filter is planned, but is not included in the current ACF. It will work on the character events and sanitize the inappropriate words. This filter will be used for blogs or SNS sites.

3.2 Architecture of JSON filtering

When the ACF filters JSON formatted strings, it uses a JSON parser to parse the strings. The JSON parser checks whether or not the JSON format in the string is correct and passes the parsed JSON value to a JSON filter. The JSON filter invokes the HTML filtering with default filter rules when the JSON value is an HTML fragment. It also strips out any "javascript:" in the JSON value when the value is not an HTML fragment (see Figure 4).

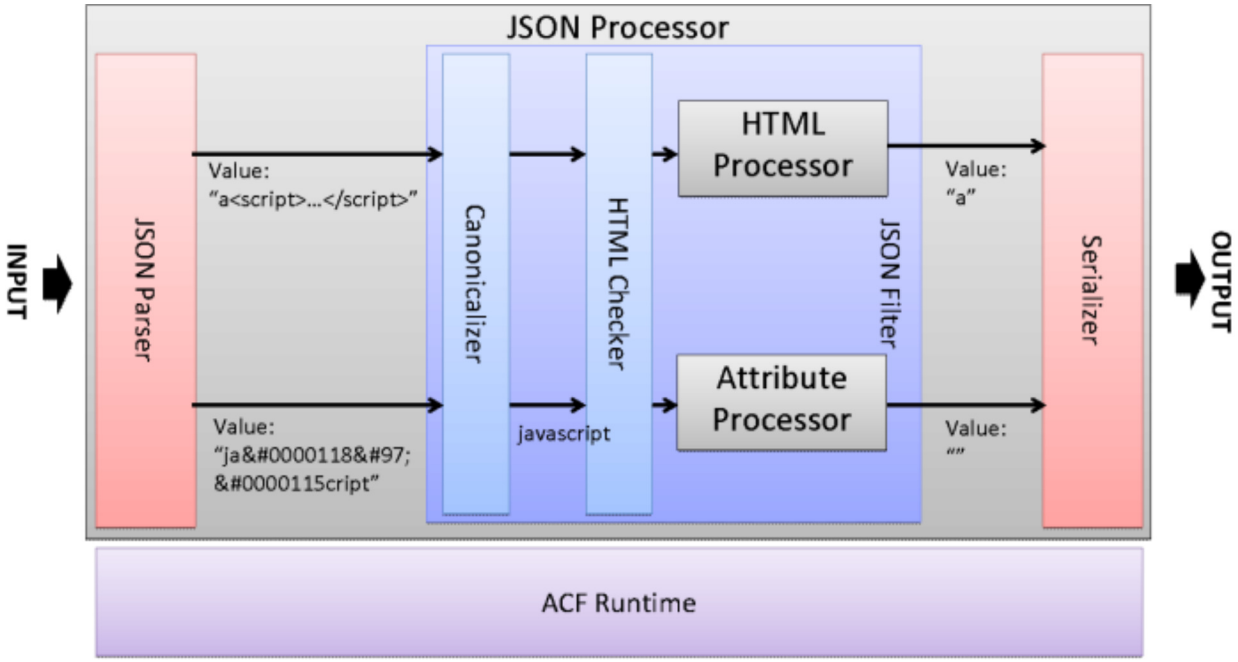

Figure 4: Architecture of JSON filtering

3.3 Architecture of RSS/ATOM feed filtering

When the ACF filters the contents of RSS/ATOM feeds, it uses a XML parser to parse the feeds. The XML parser extracts any character and any CDATA section in the RSS/ATOM feeds and passes them to a JSON filter. The JSON filter invokes the HTML filtering with default filter rules when the JSON value is an HTML fragment. It also strips out any "javascript:" in the JSON value when the value is not an HTML fragment (see Figure 5).

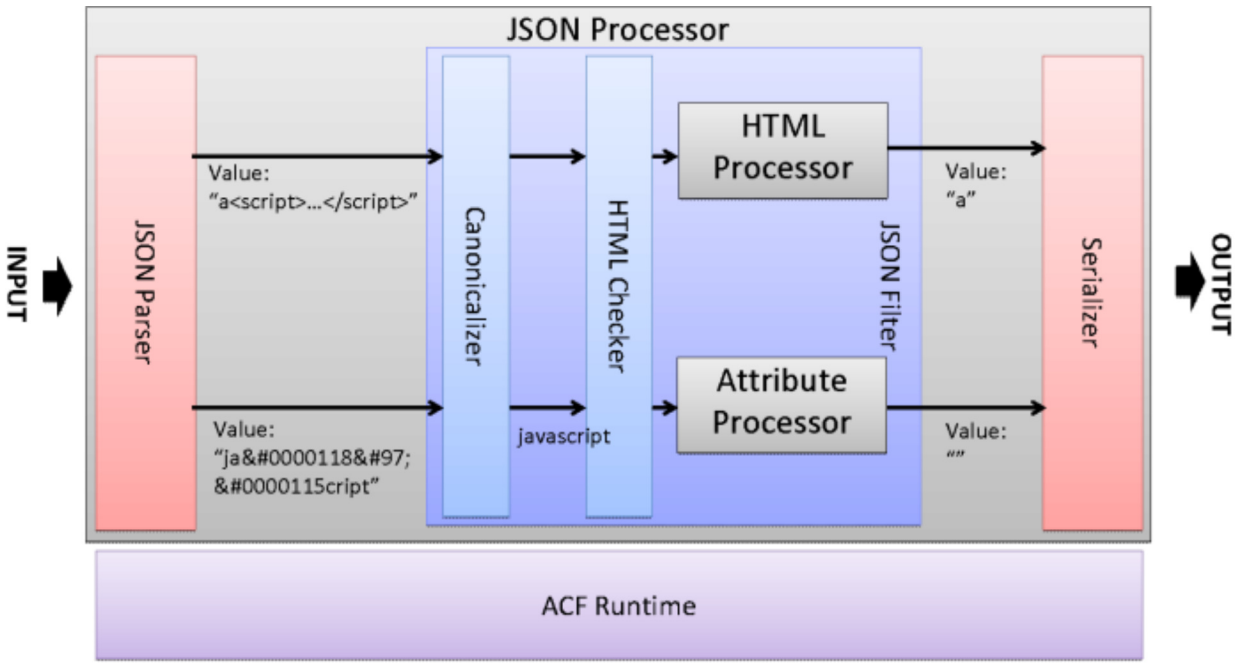

Figure 5: Architecture of JSON filtering

4. Steps to invoke the ACF

These steps invoke the ACF:

1. Write a configuration file as required.
2. Associate the content type of the input data with either HTML filtering or JSON filtering.

3. Invoke the ACF with the content type of the input data.

4.1 Editing a configuration file

The configuration file contains the data specifying the chain of filters and their filtering rules. ACF includes a default configuration file, or a custom configuration file may be created. To create a configuration files, please see the configuration file for HTML filtering section.

4.2 Associating the content type

The HTML filtering is associated with the following content types by default in the ACF.

- `"text/html"`
- `"application/xhtml+xml"`

The JSON filtering is associated with the following content types by default in the ACF.

- `"text/json"`
- `"application/json"`

The RSS/ATOM feed filtering is associated with the following content types by default in the ACF.

- `"application/rss+xml"`
- `"application/atom+xml"`
- `"application/rdf+xml"`

To associate a different content type with the HTML filtering, the JSON filtering or the RSS/ATOM feed filtering, use the `registerContentType` method in `com.ibm.trl.acf.api.ActiveContentProcessorFactory`. When a content type is associated with the HTML filtering, the processor type must be `"html"`. When the content type is associated with the JSON filtering, the processor type must be `"json"`. When the content type is associated with the RSS/ATOM feed filtering, the processor type must be `"xmlfeed"`. None of the default content types can be overridden. Here is the sample code to associate `"text/plain"` with HTML filtering. For the details, please refer to the separate ACF Javadoc.

```
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;

// 1. Get a ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFactory();

// 2. Associate "text/plain" with the HTML filtering.
factory.registerContentType("html", "text/plain");
```

4.3 Invoking the ACF

The following methods are available in the ACF factory (`com.ibm.trl.acf.api.ActiveContentProcessorFactory`) to get an ACF processor (`com.ibm.trl.acf.api.ActiveContentProcessor`). For the details of the referenced configuration files, please see the Configuration section.

- **`getActiveContentProcessor(String contentType)`**: Get the ACF processor associated with the content type. If the specified content type is associated with HTML filtering, the `"acf-xpathconfig.xml"` configuration file is used by default.
- **`getActiveContentProcessor(String contentType, String httpUserAgent)`**: Get the ACF processor associated with the content type and the HTTP user agent. Since the current implementation ignores the HTTP user agent, the behavior of this call is the same as the `getActiveContentProcessor(String contentType)`.
- **`getActiveContentProcessor(String contentType, Properties properties)`**: Get the ACF processor associated with the content type and the properties. The following properties are supported:
  - **`"acf.use.annotation"`**: Indicates if annotation-based HTML filtering is used. This property applies only with HTML filtering. If `true`, annotation-based filtering is in use, or otherwise, XPath-based filtering is used.

    - **"acf.flash.protection"**: Indicates if Flash protection is required. This property applies only with HTML filtering. If `true`, Flash protection is enabled.

  When both "acf.use.annotation" and "acf.flash.protection" are true, the "acf-annotationconfig-flash.xml" configuration file is used. When only "acf.use.annotation" is true, the "acf-annotationconfig.xml" configuration file is used. When only "acf.flash.protection" is true, the "acf-xpathconfig-flash.xml" configuration file is used. Otherwise, the "acf-xpathconfig.xml" configuration file is used.
- **getActiveContentProcessor(String contentType, String httpUserAgent, Properties properties)**: Get the ACF processor associated with the content type, the HTTP user agent, and the properties. Since the current implementation ignores the HTTP user agent, the behavior is same as the `getActiveContentProcessor(String contentType, Properties properties)`.
- **getActiveContentProcessor(String contentType, String httpUserAgent, String configKey, InputStream configFile)**: Returns the ACF processor associated with the content type, the HTTP user agent, the cache key name for the configuration file, and the input stream of the configuration file. However the HTTP user agent is ignored in the current implementation. In general, the cost of loading the configuration file is very high even though it is loaded from the input stream. Therefore specifying the cache key name as a parameter is recommended to avoid loading it internally every time.

One of the following methods is available in the ACF processor to filter an input HTML document and get the filtered document.

- **boolean process(java.io.Reader source, java.io.Writer sink, String encoding):**
- **boolean process(java.io.InputStream source, java.io.Writer sink, String encoding):**
- **boolean process(java.io.InputStream source, java.io.OutputStream sink, String encoding):**

  The ACF processor reads input from the specified source (reader or input stream) and filters it. During filtering process, the specified character encoding may be used to decode any encoding trick. The result of filtering is written into the specified sink (writer or output stream). This method returns true if there is any malicious content in the input.

One of the following methods is available in the ACF processor to validate if there is any malicious content check if there is any filter input HTML document in the input.

- **void validate(java.io.Reader source, java.io.Writer sink, String encoding):**
- **void validate(java.io.InputStream source, java.io.OutputStream sink, String encoding):**

  The ACF processor reads input from the specified source (reader or input stream) and validates it. During validating process, the specified character encoding may be used to decode any encoding trick. The result of filtering is written into the specified sink (writer or output stream) if there is no malicious content in the input. If there is any malicious content in the input, it throws an exception.

- **String validate(java.io.Reader source, String encoding):**
- **String validate(java.io.InputStream source, String encoding):**

  These two validate method can be also used. The difference between these methods and the methods above is what the ACF processor does when there is any malicious content in the input. These methods return the malicious content itself found during validation process without throwing any exception.

Here are some code samples invoking the ACF.

Example 1: Invoke annotation-based HTML filtering.

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;
import java.util.Properties;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();

// 2. Get an ActiveContentProcessor instance.
```

```
Properties properties = new Properties();
properties.put(ActiveContentProcessorFactory.PROPERTY_USE_ANNOTATION, "true");
ActiveContentProcessor processor = factory.getActiveContentProcessor("text/html",
properties);

// 3. Invoke the HTML filtering.
Reader source = ...; // Includes an HTML document or an HTML fragment
Writer sink = ...; // Used to output the result of HTML filtering
String encoding = ...; // The character encoding used in the source
processor.process(source, sink, encoding);
```

Example 2: Invoke annotation-based HTML filtering with Flash protection.

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;
import java.util.Properties;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();

// 2. Get an ActiveContentProcessor instance.
Properties properties = new Properties();
properties.put(ActiveContentProcessorFactory.PROPERTY_USE_ANNOTATION, "true");
properties.put(ActiveContentProcessorFactory.PROPERTY_FLASH_PROTECTION, "true");
ActiveContentProcessor processor = factory.getActiveContentProcessor("text/html",
properties);

// 3. Invoke the HTML filtering.
Reader source = ...; // Includes an HTML document or an HTML fragment
Writer sink = ...; // Used to output the result of HTML filtering
String encoding = ...; // The character encoding used in the source
processor.process(source, sink, encoding);
```

Example 3: Invoke the HTML filtering with a custom configuration file (e.g. `customconfig.xml`).

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;
import java.io.FileInputStream;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();

// 2. Get an ActiveContentProcessor instance.
ActiveContentProcessor processor = factory.getActiveContentProcessor("text/html", null,
"customconfig", new FileInputStream("customrconfig.xml"));

// 3. Invoke the HTML filtering.
Reader source = ...; // Includes an HTML document or an HTML fragment
Writer sink = ...; // Used to output the result of HTML filtering
String encoding = ...; // The character encoding used in the source
processor.process(source, sink, encoding);
```

Example 4: Invoke the JSON filtering.

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();
```

```
// 2. Get an ActiveContentProcessor instance.
ActiveContentProcessor processor = factory.getActiveContentProcessor("application/json"));

// 3. Invoke the JSON filtering.
Reader source = ...; // Includes a JSON-formatted string or a JSON value
Writer sink = ...; // Used to output the result of JSON filtering
String encoding = ...; // The character encoding used in the source
processor.process(source, sink, encoding);
```

Example 5: Invoke the RSS/ATOM feed filtering.

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();

// 2. Get an ActiveContentProcessor instance.
ActiveContentProcessor processor = factory.getActiveContentProcessor("application/rss+xml"));

// 3. Invoke the RSS/ATOM feed filtering.
Reader source = ...; // Includes a RSS/ATOM feed
Writer sink = ...; // Used to output the result of RSS/ATOM feed filtering
String encoding = ...; // The character encoding used in the source
processor.process(source, sink, encoding);
```

5. Configuration

The ACF is configured in a declarative way using a configuration file written in XML. Each configuration file includes only one set of ACF configuration settings. Although a sophisticated GUI can be provided for ACF configuration by other products, we will describe the low-level format of the XML file here. ACF provides the following default configuration files for the HTML filtering in the `com.ibm.trl.acf.impl.html` package. These files for annotation-based HTML filtering are loaded during the startup.

1. **`acf-annotationconfig.xml`**: ACF is configured with an HTML base filter only. In the HTML base filter, all elements and attribute values which may include malicious active content are silently removed. The default filtering scope is whole HTML document or fragment since the scope specified in this configuration file is empty.
2. **`acf-annotationconfig-flash.xml`**: ACF is configured with the HTML base filter and Flash filter. In the HTML base filter, all elements and attribute values that are not related to Flash content but that may include malicious active content are silently removed. In the Flash filter, all malicious attribute values in the elements that are related to Flash content are silently removed. In both filters, the default filtering scope is whole HTML document or fragment since the scope specified in this configuration file is empty.

In general, different application will require different configuration. ACF provides the method to use a custom configuration file. But it always forces Web application developers to specify the input stream of the custom file when they get a ActiveContentProcessor instance. In order to improve usability, ACF also provides the method to override the default configuration.

- **`ActiveContentProcessorFactory.setDefaultConfiguration(String processorType, Properties properties, InputStream configFile)`**: Load the ACF configuration from the specified input stream of a configuration file and set it as default configuration for the specified processor type and the properties. The supported processor type is `"html"`, `"json"`, and `"xmlfeed"`. The following properties are supported:
  - **`"acf.use.annotation"`**: Indicates if annotation-based HTML filtering is used. This property applies only with HTML filtering. If `true`, annotation-based filtering is in use, or otherwise, XPath-based filtering is used.
  - **`"acf.flash.protection"`**: Indicates if Flash protection is required. This property applies only with HTML filtering. If `true`, Flash protection is enabled.

The rest of this section describes general parts of the ACF configuration file.

5.1 How to write a configuration file

A configuration file is composed of two main parts. In the first part, a chain of filters is declared. In the second part, the detailed filter rules are declared for the filters that were specified in the first part (See Figure 6).

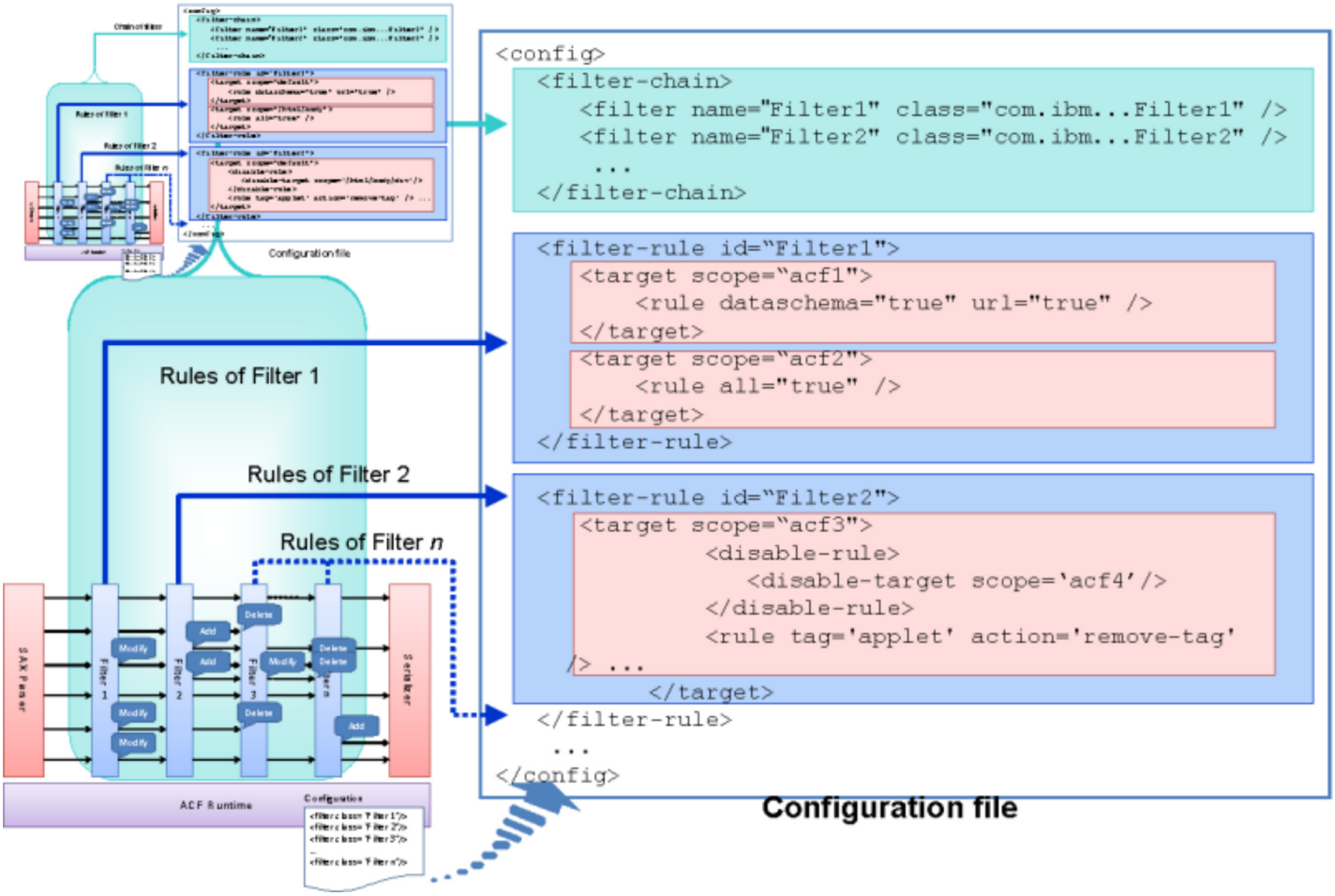

Figure 6: Composition of the configuration file

Figure 7 illustrates the structure of the configuration file.

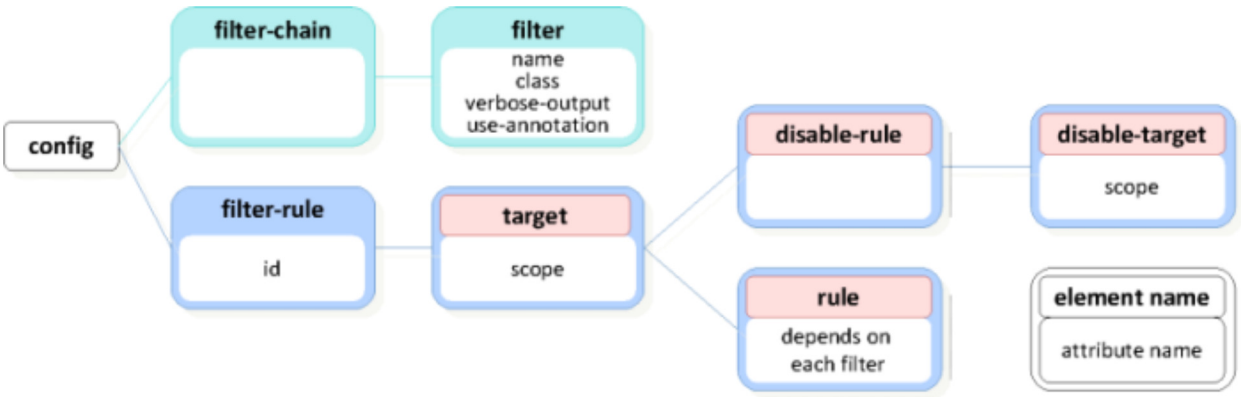

Figure 7: Structure of the configuration file

Here is the detail of the elements in the configuration file.

- **`config`**: This element is the root element of the ACF configuration file. It can be include only one filter chain.
- **`filterChain`**: This element contains a chain of filters. Each filter, described as a child element of this element, is applied in the order of appearance.

- **`filter`**: This element contains the basic information for a filter. This element can appear 0 (zero) or more times in the configuration file. The attributes of this element are:
  - **`@name`** (Required): An arbitrary name for the filter.
  - **`@class`** (Required): A name for the class that extends the `com.ibm.trl.acf.api.Filter` abstract class.
  - **`@verbose-output`** (Optional): The default value is `false`. If it is set to `true`, a comment is inserted whenever active content is removed.
  - **`@use-annotation`** (Optional): The default value is `true`. This attribute is used only for the HTML filtering. If it is set to `true`, annotation is used to limit the scope of the HTML filtering. Otherwise, XPath is used to limit the scope.

  Here is an example of a `filter` element:

```
<filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter" verbose-
output="true" use-annotation="false">
```

- **`filter-rule`**: The actual filtering rule for each filter declared in the <filter> elements in the <filter-chain>. Each filter chain is required to have at least one filter rule. The attribute of this element is:
  - **`@id`** (Required): The associated identifier of the filter rule that is specified in the <filter> as `filter/@name`.

  Here is an example of a `filter-rule` element:

```
<filter-rule id="base">
```

- **`target`**: This element specifies the filtering scope, and may appear one or more times. Here is the attribute of this element:
  - **`@scope`** (Required): When annotation-based HTML filtering is used, this attribute value is a prefix for the annotations used to limit the filtering scope. When the value is empty, the rules are applied to all of the input. When using XPath-based HTML filtering, this attribute is an XPath expression used to limit the filtering scope. In that case, when the value is empty or "default", the rules are applied to all of the input.

  Here is an example of a `target` element:

```
<target scope="acfscope1">
```

- **`rule`**: This element contains the parameters of the filter as custom attributes. The attributes that are defined depend upon the filter being configured.

  Here is an example of a `rule` element:

```
<rule schema="true" url="false"> <!-- for the attribute value canonicalizer -->
<rule tag="script" action="remove-tag"> <!-- for the HTML base filter -->
```

For example, suppose the following configuration file for annotation-based HTML filtering.

```
<config>
   <filter-chain>
      <filter name="Filter" class="...">
   </filter-chain>
   <filter-rule id="Filter">
      <target scope="acf1">
         <rule .../>
         <rule .../>
          ...
      </target>
      <target scope="acf2">
         <rule .../>
          ...
      </target>
   </filter-rule>
</config>
```

In this case, the ACF filters the input HTML document as shown in Figure 8.

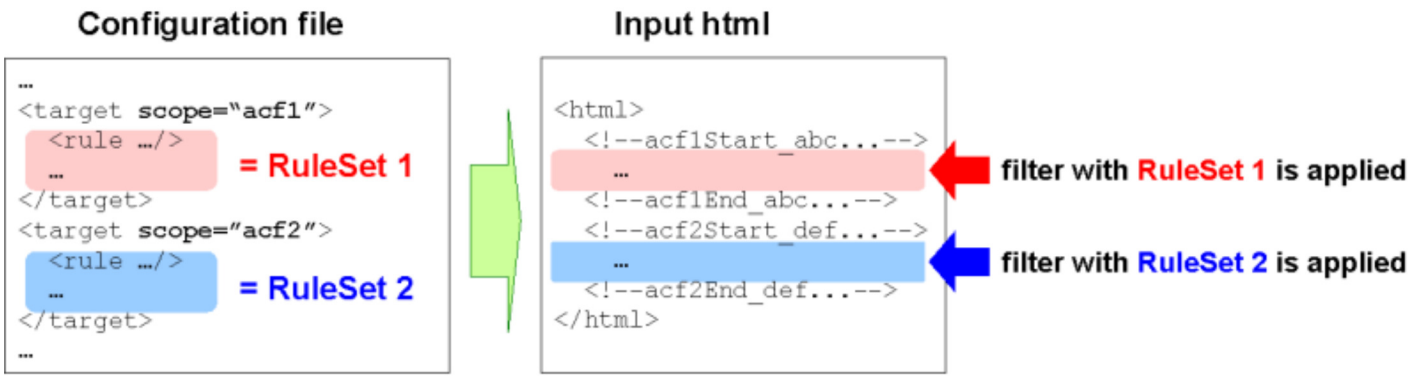

Figure 8: An example of the filter behavior for annotation-based HTML filtering

5.2 DTD of the ACF configuration file

The DTD of the ACF configuration file is as follows.

```
<!ELEMENT config (filter-chain, filter-rule*) >

<!ELEMENT filter-chain (filter+) >

<!ELEMENT filter EMPTY >
<!ATTLIST filter name ID #REQUIRED
                 class CDATA #REQUIRED
                 verbose-output (true|false) "false"
                 use-annotation (true|false) "true" >

<!ELEMENT filter-rule (target+) >
<!ATTLIST filter-rule id IDREF #REQUIRED >

<!ELEMENT target (rule*) >
<!ATTLIST target scope CDATA "default" >

<!ELEMENT rule EMPTY >
```

6. Default filter implementations

6.1 HTML filtering

The ACF provides the following three default filters for the HTML filtering: Attribute value canonicalizer, HTML base filter, and Flash filter.

6.1.1 Attribute value canonicalizer

Some security reports have reported various encoding tricks in attribute values are being used to bypass security filtering. Attackers can try to bypass ACF by using these encoding tricks. In order to protect against the encoding tricks, the attribute value canonicalizer can provide the following functions for all attribute values:

- Decodes any attribute values that are encoded with the data URL scheme (RFC2397). Here is an example of an encoding based on the data URL scheme:

```
data:text/html;base64,amF2YXNjcmlwdDphbGVydCgnbWFsJyk7=
-->
javascript:alert('mal');
```

- Resolve any entity references in an attribute value. Here is an example of an entity reference:

```
&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;
(&#39;&#109;&#97;&#108;&#39;)&#59;
-->
javascript:alert('mal');
```

- Decode any URL encoding in an attribute value. Here is an example of a URL encoding:

```
J%97vasc%72ipt%3Aalert%28%27mal%27%29%3B
-->
javascript:alert('mal');
```

- Remove any white space (such as tabs, carriage returns, and line feeds). Here is an example of white spaces

```
jav
ascr
ipt:alert('mal');
-->
javascript:alert('mal');
```

When IE v6 is used in client's environment, the following encoding tricks to obfuscate style attribute need to be considered (Please see Cause of XSS by excessive detection of "expression" in IE for more details):

- Inject any comment:

```
<div style="{ left:exp/*  */ression( alert('xss') ) }">
-->
<div style="{ left:expression( alert('xss') ) }">
```

- Use any fullwidth letter:

```
<div style="{
left:&#xFF25;&#xFF58;&#xFF30;&#xFF52;&#xFF25;&#xFF53;&#xFF33;&#xFF49;&#xFF2F;&#xFF4E;(
alert('xss') ) }">
-->
<div style="{ left:expression( alert('xss') ) }">
```

- Use any particular Unicode character:

```
<div style="{ left:ex&#x0280;ressio&#x207F;( alert('xss') ) }">
-->
<div style="{ left:expression( alert('xss') ) }">
The attribute value canonicalier can also prevent such encoding tricks.
```

Configuration

The attribute value canonicalizer is enabled in the ACF configuration for decoding the attribute values when the Flash filter `"com.ibm.trl.acf.impl.html.canonicalizer.C14NFilter"` is included. In addition, the following rule must be included to enable the attribute value canonicalizer.

```
<rule all="true" />
```

To decode encodings based on the data URL scheme or URL encoding, the following rule can be used instead. However this rule does not resolve any entity references and it does not remove any white space.

```
<rule dataschema="true" entityreference="true" url="false" whitespace="false"/>
```

Here is an example of an ACF configuration file:

```
<config>
   <filter-chain>
      <filter name="c14n" class="com.ibm.trl.acf.impl.html.canonicalizer.C14NFilter"
use-annotation="false" />
   </filter-chain>
   <filter-rule id="c14n">
      <target scope="">
         <rule all="true" />
      </target>
   </filter-rule>
</config>
```

6.1.2 HTML base filter

To remove elements and attribute values which satisfy the conditions specified in the ACF configuration file, use XPath-based HTML filtering (deprecated) or annotation-based HTML filtering.

Annotation-based HTML filtering: There is a fundamental problem in XPath-based HTML filtering. For example, if the XPath `"/html/body//div"` is used in the configuration file, then certain malicious code can escape filtering. When the HTML document includes the following user input,

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>
            <!-- user input here -->
            hello<br>
            </table>
            <script>alert('xss')</script>
          </div>
        </td>
      </tr>
    <table>
  </body>
</html>
```

the filtering fails to remove the suspicious HTML (the script with the alert). The filtering scope is shown in red, because there are some missing parent end tags such as `</div>`, `</td>`, and `</tr>`.

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>
            <!-- user input here -->
            hello<br>
            </div></td></tr></table>
            <script>alert('xss')</script>



  </body>
</html>
```

The malicious script tag is not removed because it is outside the scope of the filtering. In this case, annotation-based HTML filtering will remove the script tag when `"config/filter-chain/filter@use-annotation"` is set to `"true"` for the HTML base filter.

To use annotation-based HTML filtering, the annotations generated by `com.ibm.trl.acf.api.ActiveContentProcessorFactory` must be inserted into the HTML document. The format of the annotation is as follows: a prefix + "Start" or "End" + &quot_" + a Base64 encoded secure random number. When the prefix is `"acf1"` and `"Start"` is used, the annotation becomes:

```
<!--acf1Start_abc***-->
```

When the prefix is `"acf2"` and `"End"` is used, the annotation becomes:

```
<!--acf2End_def***-->
```

The secure random number affects the strength of the security, so it is better to generate a random number for each request. However the number will be is cached in the `ActiveContentProcessorFactory` for a specified duration because generating a new random number at each step is too time consuming. To control the use of the random number, the following parameters should be used:

- Caching duration: The time period for which the number is cached in the `ActiveContentProcessorFactory`. The default duration is one day (86,400 seconds).
- Size of the number: The size of the number. The default size is 20 bytes (160 bits).

The `ActiveContentProcessorFactory` uses the following eight methods to generate annotations:

- **`getStartAnnotation()`**: Get a start annotation generated with the default prefix "default" and the default size of the secure random number. The annotation indicates the start point of the HTML filtering and the number is cached for the default duration.
- **`getStartAnnotation(String prefix)`**: Get a start annotation generated with the specified prefix and the default size of the secure random number. The annotation indicates the start point of the HTML filtering and the number is cached for the default duration.
- **`getStartAnnotation(int duration, int length)`**: Get a start annotation generated with the default prefix "default" and the specified size of the secure random number. The annotation indicates the start point of the HTML filtering and the number is cached for the specified duration.
- **`getStartAnnotation(String prefix, int duration, int length)`**: Get a start annotation generated with the specified prefix and the specified size of the secure random number. The annotation indicates the start point of the HTML filtering and the number is cached for the specified duration.
- **`getEndAnnotation()`**: Get an end annotation generated with the default prefix "default" and the default size of the secure random number. The annotation indicates the end point of the HTML filtering and the number is cached for the default duration.
- **`getEndAnnotation(String prefix)`**: Get an end annotation generated with the specified prefix and the default size of the secure random number. The annotation indicates the end point of the HTML filtering and the number is cached for the default duration.
- **`getEndAnnotation(int duration, int length)`**: Get an end annotation generated with the default prefix "default" and the specified size of the secure random number. The annotation indicates the end point of the HTML filtering and the number is cached for the specified duration.
- **`getEndAnnotation(String prefix, int duration, int length)`**: Get an end annotation generated with the specified prefix and the specified size of the secure random number. The annotation indicates the end point of the HTML filtering and the number is cached for the specified duration.

The generated secure random numbers are managed in the `ActiveContentProcessorFactory` based on the combination of the prefix of the annotations, the caching duration of the secure random number, and the length of the secure random number. Suppose the annotation is `a1`, the time when the annotation was generated is `0`, the caching duration is `d`, and the current time is `ct`. When $0 \leq ct \leq d$, then `a1` is active. When $d < ct \leq 2d$, then a new annotation `a2` is generated but `a1` is still active. When $2d < ct$, then `a1` has expired. Please see Figure 9.

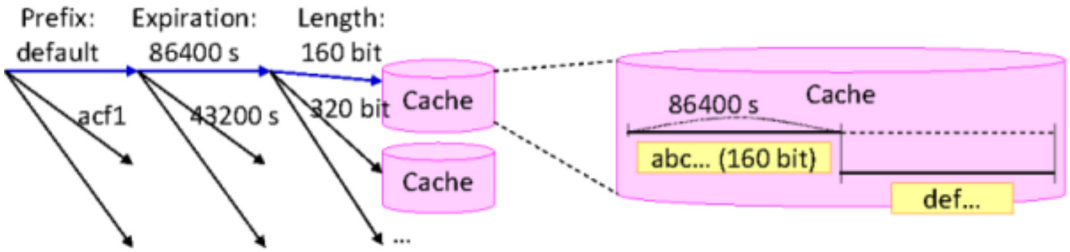

Figure 9: Management of annotations

If the ACF receives the following HTML document (which XPath-based filtering cannot handle correctly), annotation-based filtering can filter the parts in red.

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>
            <!--acf1Start_abc***-->
            <!-- user input here -->
            hello<br>
            </table>
            <script>alert('xss')</script>
            <!--acf1End_abc***-->
          </div>
        </td>
      </tr>
    <table>
  </body>
</html>
```

Then it outputs the following expected result.

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>

            <!-- user input here -->
            hello<br>
            </div></td></tr></table>




  </body>
</html>
```

Configuration

The HTML base filter in the ACF configuration is enabled when the HTML base filter named `"com.ibm.trl.acf.impl.html.basefilter.BaseFilter"`is included. The HTML base filter can be used with the attribute value canonicalizer, but the original information about the input HTML document will be lost. For example, if the ACF receives the following input,

```
<DIV ID="&#106;&#97;&#118;&#97;1">
  <IMG SRC="http://...">
  <BR>
  ...
```

```
</DIV>
```

then the result of the ACF processing appears as below when both the attribute value canonicalizer and the HTML base filter are used.

```
<div id="java1">
  <img src="">
  <br>
  ...
</div>
```

To retain the original information, it is necessary to use the HTML base filter only. However including the following rule allows using the same functions as the attribute value canonicalizer.

```
<rule c14n="true" all="true" />
```

In this case, here are the results of the ACF processing:

```
<DIV ID="&#106;&#97;&#118;&#97;1">
  <IMG SRC="">
  <BR>
  ...
</DIV>
```

Please see the configuration for the attribute value canonicalizer section for the details.

The HTML base filter rule element is specified in the form of:

```
<rule tag="tag_name" attribute="attribute_name" value="keywords" value-criterion="criterion"
action="action" />
```

The meaning of each attribute is:

- **`tag`**: The name of the HTML element for which the filter is invoked.
- **`attribute`**: The name of an attribute in the HTML element.
- **`value`**: When the keyword specified by the value attribute matches the attribute value, then the action is taken.
- **`value-criterion`**: The criterion of the keyword match. The criterion can be "equals", "starts-with", or "contains".
- **`action`**: The action to be taken when the keyword matches. The action can be "remove-tag", "remove-attribute-value", "remove-tag-as-html", "remove-attribute-value-as-html", or "remove-attribute-value-as-css". "remove-tag" is used just to remve a tag. "remove-attribute-value" is used just to remove a attribute value. It is basically enough to use these two actions. However ACF supports some special actions for the specific cases. "remove-tag-as-html" can be used to remove a tag when an attribute value in the tag includes an HTML fragment which has any malicious content. For example, it is useful to remove <META> tag if its 'content' attribute includes a malicious HTML fragment. "remove-attribute-value-as-html" can be used to remove an attribute value when the value includes an HTML fragment which has any malicious content. For example, it is useful to remove 'src' attribute which includes Base64 encoded <SVG> tag, like <EMBED src="..." ...>. "remove-attribute-value-as-css" can be used to remove the value of any style attribute.

Here are sample configuration files for annotation-based HTML filtering.

Example 1: The HTML base filter with a rule that removes script tags in the input, which is applied only between annotations which start with the prefix "acf1".

```
<config>
   <filter-chain>
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter"
```

```
use-annotation="true" />
   </filter-chain>

   <filter-rule id="base">
      <target scope="acf1">
         <rule c14n="true" all="true" />
         <rule tag="script" action="remove-tag" />
      </target>
   </filter-rule>
</config>
```

The results of this configuration appear in Figure 10.

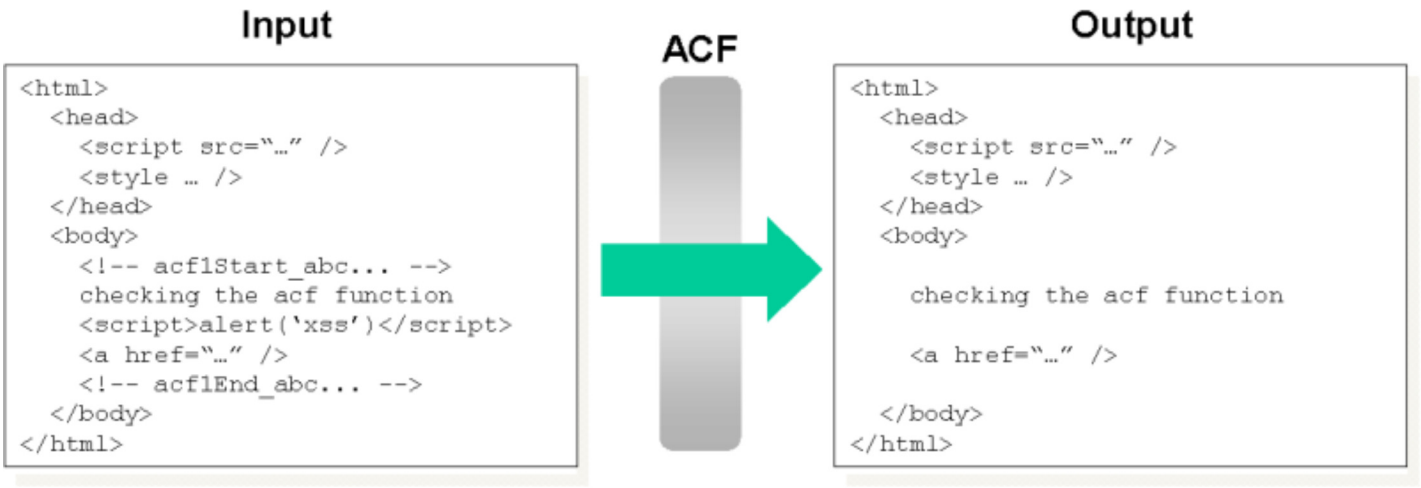

Figure 10: Input and output when the configuration "Example 1" for annotation-based HTML filtering is used

Example 2: The HTML base filter with two targets is applied. Both script tags and iframe tags are removed only between annotations which start with the prefix "acf1". Only script tags are removed between annotations which start with the prefix "acf2".

```
<config>
   <filter-chain>
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter" />
   </filter-chain>

   <filter-rule id="base">
      <target scope="acf1">
         <rule tag="script" action="remove-tag" />
         <rule tag="iframe" action="remove-tag" />
      </target>
      <target scope="acf2">
         <rule tag="script" action="remove-tag" />
      </target>
   </filter-rule>
</config>
```

The results of this configuration appear in Figure 11.

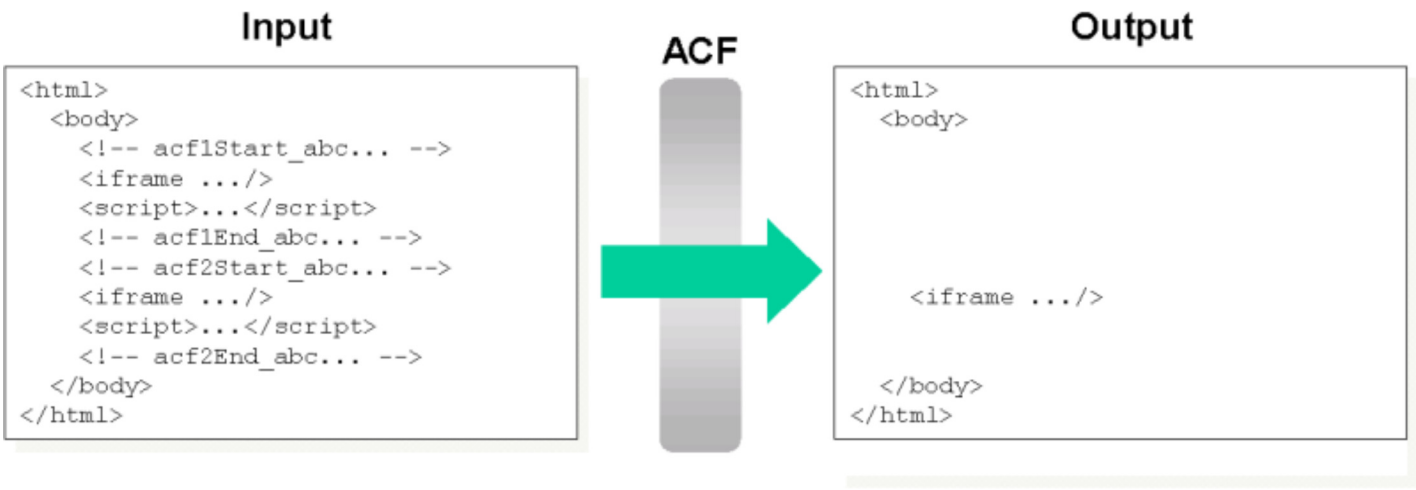

Figure 11: Input and output when the configuration "Example 2" for annotation-based HTML filtering is used

How to make a filter?

In addition to the default filters (Attribute value canonicalizer, HTML base filter, and Flash filter), custom filters can be created. There are two steps for creating and using a new filter:

1. Implement a custom filter by extending the `com.ibm.trl.acf.api.html.Filter` class. The `Filter` class defines three abstract methods that must be implemented: `startElement2`, `endElement2`, and `characters2`. These methods have the same semantics as `startElement`, `endElement`, and `characters` in the `org.xm.sax.ContentHandler`. The difference is that they apply only to the scope declared in the configuration file. For example, here are the steps to create `"MyFilter"`:
    1. Create a `MyFilter` class by extending the `Filter` class.
    2. Call the constructor of the superclass in the constructor.
    3. Implement the actual process when each method is called.
        - The interface is almost the same as for `org.xm.sax.ContentHandler`. Instead of `startElement`, `endElement`, and `characters`, the methods `startElement2`, `endElement2`, and `characters2` should be implemented.
        - The difference between `startElement` and `startElement2` is that `startElement2` has an additional argument for `"applicableRules"`. This argument contains a list of the applicable rules specified in the configuration file. Rules that are not in the current scope will be ignored.
        - Unchanged methods do not need to be implemented.

    The example code is shown below.

```
package myfilter;

import com.ibm.trl.acf.api.html.Filter;
import com.ibm.trl.acf.api.html.FilterRule;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;

public class MyFilter extends Filter {
    public MyFilter(
            AbstractFilter next, FilterRule filterRule, Boolean isProcess, String encoding
    ) throws ActiveContentException {
      // Call the constructor of the superclass
      super(next, filterRule, isProcess);
   }

    public void startElement2(
            String uri, String localName, String qName, Attributes attrs,
```

```
            List applicableRules
    ) throws SAXException {
        if (logger.isLoggable(Level.FINER)) {
            logger.entering(
                    CLS, "startElement2",
                    new Object[] { uri, localName, qName, attrs, applicableRules }
            );
        }

        // Write the actual process for when a start element arrives.
        //
        // Note: How to throw a SAX event to the next handler is:
        //
        // if (next != null) {
        //     next.startElement(uri, localName, qName, newAttributeList);
        // }

        if (logger.isLoggable(Level.FINER)) {
            logger.exiting(CLS, "startElement2");
        }
    }

}
```

2. 2. Add the filter rules to the ACF configuration file. How to write the configuration file is described in the Configuration section. Custom attribute value pairs to configure the filter can be written among the <rule> elements in the configuration file. Those values can then be accessed through the `applicableRules` attribute of `startElement2`.

6.1.3 Flash filter

Recently detected attacks exploit the flexible functionality of Flash objects (SWF) or ActionScript. These attacks can be classified into two categories: a) attacks by malicious SWF, and b) attacks by malicious HTML strings that exploit a vulnerability in an innocent SWF object.

Attack Pattern A: Malicious SWF attacking the HTML content

1. 1. SWF can execute malicious JavaScript by using the ActionScript API. Some ActionScript API calls take a URL as a parameter, and allow an SWF object to access a remote resource at the specified URL. However, such and API may also take a `"javascript:"` URI that is followed by a piece of JavaScript code. For example, an attacker can execute JavaScript by using the getURL function as follows:

   ```
   getURL("javascript:alert('xss')", "_self");
   ```

   Some other functions, such as various load functions have similar capabilities.

   ```
   loadVariables("javascript:alert('xss')", level);
   loadMovie("javascript:alert('xss')", target);
   XML.Load("javascript:alert('xss')");
   ```

2. SWF can generate a subset of HTML using ActionScript, which can include JavaScript code to be invoked. Here is an example with ActionScript:

   ```
   textfield.html = true;
   textfield.htmlText = "<a href="javascript:alert('xss')">Click here!</a>";
   ```

3. The `ExternalInterface` API in ActionScript allows an SWF object to call a JavaScript function in the browser, and an attacker can execute arbitrary code by using this function. For example, `ExternalInterface` can call a JavaScript function and pass any values.

```
ExternalInterface.call("window.open", "javascript:alert('xss')");
```

4. The `System.fscommand()` API allows ActionScript to invoke a JavaScript call-back function defined in the parent HTML. For example, suppose a callback function is defined in JavaScript in the parent HTML:

```
function myMovie_DoFSCommand (arg1, arg2){ // arg1 is a URL
   window.open(arg1, arg2)
};
```

Then ActionScript can call it as follows:

```
System.fscommand("javascript:alert('xss')", "he he he");
```

Attack Pattern B: Malicious HTML exploiting vulnerability in an SWF object for an XSS attack

Even innocent SWF objects may have some vulnerability that can be exploited by a malicious HTML document.

1. Script injection through URL request parameters and `FlashVars`: When embedding an SWF object into an HTML file, the parent HTML can pass parameters to the SWF through certain HTML attributes. For example, when an SWF object is inserted in the HTML by using the `<embed>` element, the URL of the SWF object can be followed by a set of request parameters, and then these parameters can be retrieved by the ActionScript in the SWF. Some Flash objects are deliberately designed to take parameters from the request parameter, to allow customization of the behavior without modifying the SWF itself. For example, an SWF can be inserted in an HTML file as:

```
<embed src="test.swf?par1=http://foo.bar.com/">
```

Then the `test.swf` object can retrieve the parameter and use it to jump to a specific URL:

```
getURL(par1);
```

However, an attacker may pass a `"javascript:"` URL instead of an ordinary HTTP URL and then the `test.swf` object can retrieve the parameter and use it to jump to a specific URL:

```
<embed src="test.swf?par1=javascript:alert('xss');&par2=xyz">
```

This results in a successful XSS attack. In addition, malicious script may be injected into uninitialized global variables. For example, the following ActionScript code checks the whether the variable `"language"` is initialized, and uses it as part of the URL from which content is loaded:

```
if (_root.language != undefined) {
    Locale.DEFAULT_LANG = _root.language;
}
obj.load(Locale.DEFAULT_LANG + '/player_' + Locale.DEFAULT_LANG + '.xml');
```

However, an attacker may override the variable `"language"` with a request parameter:

```
http://foo.bar.com/?language=javascript:alert('xss')
```

FlashVars is another mechanism to pass a set of parameters in the form of name and value pairs. For example, the following HTML fragment executes the same attack as used in the previous example:

```
<embed src="test.swf" FlashVars="par1=javascript:alert('xss');&par2=xyz">
```

Another example of injection is to insert a piece of ActionScript code with an `"asfunction:"` URL. The `"asfunction:"` URL is similar to the `"javascript:"` URL. When a URL `"asfunction:x,y"` is accessed by ActionScript, it invokes the function x with the parameter y. For example:

```
<embed src="test.swf" FlashVars="par1=asfunction:getURL,javascript:alert('xss')" ...>
```

In this example, when ActionScript accesses the URL of `par1`, it will invoke the getURL function with the parameter `"javascript:alert('xss')"`, which results in the invocation of the JavaScript. Table 1 shows HTML elements and attributes that can be used to pass values to SWF objects.

Table 1: HTML elements and attributes for Flash

Element	Attribute of SWF URL	Attribute of parameters	Description
object	data	FlashVars	An object element may include nested embed or object elements.
embed	src	FlashVars	
iframe	src	FlashVars	SWF is loaded in the iframe
frame	src	FlashVars	SWF is loaded in the frame
a	src	FlashVars	The browser navigates to the URL when the link is clicked on
img	src	FlashVars	
Param	Value (<param name="movie" value="swf_url">)	Value (<param name="flashvars" value="name_value_pairs">)	<param> may be used within an object element. When name="movie", the value attribute includes a SWF URL. When name="flashvars", the value attribute includes parameters in the form of FlashVars.

2. Several JavaScript libraries, such as SWFObject 1.5, SWFObject 2.0, and UFO Open Source Flash allow a Web developer to insert an SWF file into the HTML. Such libraries are convenient because they mask the difference between browser and Flash player versions. The attacks described in 1 (the previous entry) are also possible through these libraries. For example, this SWFObject has an effect identical to the attack in 1:

```
swfobject.embedSWF("test.swf?par1=javascript:alert('xss')", ...);
```

3. Data passing by `SetVariable` and `GetVariable`: JavaScript can read data from Flash by using the `GetVariable` function, which allows JavaScript to access public and static objects in an SWF object. For example:

```
document.myMovie.GetVariable("_level0.important");
```

The `SetVariable` function allows JavaScript to write into a static or public object in an SWF object. For example:

```
document.myMovie.SetVariable("URL","javascript:alert('xss');");
```

An attacker can use these functions to inject scripts into Flash.

4. ActionScript callback function: JavaScript can execute an ActionScript callback function defined in Flash. For example, an ActionScript callback function can be defined in Flash as:

```
ExternalInterface.addCallback("myFunction", ...)
```

Then JavaScript can execute this function as:

```
document.myMovie.myFunction("");
```

An attacker can use these functions to inject scripts into Flash.

Preventing Flash-XSS with ACF

To prevent Flash-XSS, ACF provides several features. All of the current features filter out and rewrite the HTML content to mitigate the risks of Flash-XSS attacks. In other words, the content type that is filtered by ACF is `"text/html"`, not `"application/x-shockwave-flash"`. Note that ACF currently does not support the filtering of `"application/x-shockwave-flash"`. The Flash-XSS prevention uses one or more of the following approaches:

1. Removal of malicious request parameters and FlashVars (corresponds to the Attack Pattern B). There are two approaches:
    - Using the base HTML filter for additional rules to remove malicious request parameters and FlashVars.
    - Using the Flash filter to remove malicious request parameters and FlashVars.

    The difference between a) and b) is that in a), all of the URL or FlashVars are completely removed if any request parameter includes potentially malicious JavaScript. In b), only the specified parameter is removed and the rest of the URL is preserved. For example, if the original HTML is:

    ```
    <embed src="test.swf" FlashVars="par1=javascript:alert('mal');&par2=xyz">
    ```

    The result of a) is: `<embed src="test.swf" FlashVars="">`.
    The result of b) is: `<embed src="test.swf" FlashVars="par2=xyz">`.

2. Rewriting HTML elements (such as `<embed>`, `<object>`, and `<img>`) in the `<iframe>` element can protect HTML content by using the browser's same-origin policy. The latest versions of the Flash players implement the same-origin policy, and prevent SWF objects from interacting via HTML if they come from different domains. However, the same-origin policy depends on the version of Flash. By rewriting some HTML elements into an `<iframe>`, we can mitigate the effect of SWF by leveraging the browser's sandbox capability, and thereby confine an objects behavior within an iframe of the browser. For example, if the original HTML is::

    ```
    <OBJECT classid="..." ...>
    <PARAM NAME=movie VALUE="fi1.swf">
    <PARAM NAME=loop VALUE=true>
    ...
    </OBJECT>
    ```

    The result of rewriting is:

    ```
    <iframe src="fi1.swf" loop="true" .../>
    ```

3. Overriding JavaScript libraries such as `SWFObject` allowing verifying and sanitizing the SWF URLs and FlashVars parameters that are passed to those libraries.

Configuration

The Flash-XSS prevention features in the ACF can be enabled as follows:

1. Removal of malicious request parameters and FlashVars (corresponds to the Attack Pattern B). Choose one of the following:

a. Use the base HTML filter with additional rules to remove malicious request parameters and FlashVars. This can be done by adding the following rules into the ACF configuration file:

```
<rule tag="param" attribute="value" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="embed" attribute="src" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="embed" attribute="FlashVars" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="embed" attribute="pluginspage" value="javascript:" value-
criterion="contains" action="remove-attribute-value" />
<rule tag="iframe" attribute="src" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="iframe" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-attribute-value" />
<rule tag="frame" attribute="src" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="frame" attribute="FlashVars" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="a" attribute="href" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="a" attribute="FlashVars" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="img" attribute="src" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="img" attribute="FlashVars" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="object" attribute="data" value="javascript:" value-criterion="contains"
action="remove-attribute-value" />
<rule tag="object" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-attribute-value" />
```

b. Use the Flash filter to remove malicious request parameters and FlashVars. The configuration needs to include the Flash filter with the class name `"com.ibm.trl.acf.impl.html.flashfilter.FlashFilter"`, and provide additional rules to remove the parameters. The Flash filter rule element is specified in the form of:

```
<rule tag="tag_name" attribute="attribute_name" value="keywords" value-
criterion="criterion" action="action" />
```

The meaning of each attribute is:

- **`tag`**: The name of the HTML element for which the filter is invoked.
- **`attribute`**: The name of an attribute in the HTML element.
- **`value`**: When the keyword specified by the value attribute matches the attribute value, then the action is taken.
- **`value-criterion`**: The criterion of the keyword match. The criterion can be "equals", "starts-with" or "contains".
- **`action`**: The action to be taken when the keyword matches. The action can be "remove-value".

Here is an example configuration:

```
<config>
  <filter-chain>
    <filter name="flash" class="com.ibm.trl.acf.impl.html.flashfilter.FlashFilter" />
  </filter-chain>

  <filter-rule id="flash">
    <target scope="">
      <rule c14n="true" all="true" />
      <rule tag="param" attribute="value" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="embed" attribute="src" value="javascript:" value-criterion="contains"
action="remove-value" />
      <rule tag="embed" attribute="FlashVars" value="javascript:" value-
```

```
criterion="contains" action="remove-value" />
      <rule tag="embed" attribute="pluginspage" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="iframe" attribute="src" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="iframe" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="frame" attribute="src" value="javascript:" value-criterion="contains"
action="remove-value" />
      <rule tag="frame" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="a" attribute="href" value="javascript:" value-criterion="contains"
action="remove-value" />
      <rule tag="a" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="img" attribute="src" value="javascript:" value-criterion="contains"
action="remove-value" />
      <rule tag="img" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="object" attribute="data" value="javascript:" value-
criterion="contains" action="remove-value" />
      <rule tag="object" attribute="FlashVars" value="javascript:" value-
criterion="contains" action="remove-value" />
    </target>
  </filter-rule>
</config>
```

2. Rewrite the HTML elements (such as `<embed>`, `<object>`, and `<img>`) in the `<iframe>` element. The configuration needs to include the Flash HTML filter with the class name `"com.ibm.trl.acf.impl.html.flashfilter.FlashFilter"`, and provide additional rules to remove the parameters. In addition, the following rule has to be included to enable HTML element rewriting for all types of elements that can insert Flash (i.e., all of the embed, object, and img elements will be rewritten):

```
<rule rewrite="true" all="true" />
```

To rewrite only some of the elements, the following rule can be used instead. This rule rewrites only `<embed>` and object but not `<img>` elements.

```
<rule rewrite="true" embed="true" object="true" img="false"/>
```

The rewrite rule can be used in an extended form allowing rewriting of all `<object>` and `<embed>` but with whitelisting certain attributes. On the `<object>` the `classid` attribute can be whitelisted. On the `<embed>` the `type` and the `pluginspage` can be whitelisted. The whitelisted attributes will be evaluated using a logical OR so if either of them appear the element will not be rewritten. Multiple entries for those attributes are separated by a space. This whitelisting capability can introduce potential security problems if harmful elements are whitelisted. It therefore needs to be used with extreme care.

```
<rule rewrite='true' all='true'
  allowed-classid="classid_1 classid_2 classid_3"
  allowed-pluginspage="pluginspage_url_1 pluginspage_url_2 pluginspage_url_3"
  allowed-type="type_1 type_2 type_3"
/>
```

3. Overriding JavaScript libraries. The JavaScript library, `"acf4swfobject.js"` overrides popular JavaScript libraries that are used to insert SWF objects into an HTML document, and sanitizes URLs and FlashVars parameters before they are passed to the actual libraries. In order to activate this feature, place `"acf4swfobject.js"` on the Web server, and then insert a reference to `"acf4swfobject.js"` into the HTML document as follows:

```
<html>
  <head>
    <script type="text/javascript" src="swfobject.js"></script>
```

```
    <script type="text/javascript" src=" .../acf4swfobject.js"></script>

    <script type="text/javascript">
      // embed Flash by using SWFObject 2.0
      swfobject.embedSWF("test.swf?clickTag=javascript:alert('XSS');", "myContent1", "400",
"300", "9.0.0");
    </script>
c
```

This example assumes that `"swfobject.js"` is the URL of the `SWFObject` JavaScript library. The `<script>` element for `<acf4swfobject.js>` must be inserted after `<swfobject.js>`, and before the `SWFObject` library function is called. In the current implementation, `SWFObject` versions 1.5 and 2, as well as UFO, are supported. Note: `<acf4swfobject.js>` should be used only when you are sure that there is a reliable JavaScript program that calls `SWFObject` or UFO, but there is a possibility of malicious URLs and FlashVars parameters being passed to the `SWFObject`. For example, a typical server side program might be written as:

```
swfobject.embedSWF("<%=swfurl%>", "myContent1");
```

In this example, `swfurl` can come from external user input. In such a case, the generated content would look like:

```
swfobject.embedSWF("tuto.swf?clickTag=javascript:alert('xss');", "myContent1");
```

This can happen when `swfurl` includes an injected script. However, an attacker could also carry out XSS in the traditional way, in which `swfurl` could include a double-quote character to disguise the end of the string literal, followed by the insertion of an additional malicious script, such as:

```
swfobject.embedSWF("tuto.swf", "myContent1"); alert("xss"); /* or do anything malicious
here */ dummyfunc("a", "myContent1");
```

In order to avoid these attacks, the server side application should make sure that special characters in `swfurl`, such as double-quotes characters, are properly sanitized or escaped when the HTML document is generated.

6.2 JSON filtering

The ACF provides a JSON filter for JSON filtering.

6.2.1 JSON filter

When the JSON processor receives a JSON-formatted string such as `{"p1", "http://www.ibm.com/***"}`, it invokes a JSON parser to parse the string. While parsing the string, the JSON parser passes each JSON value to the JSON filter. When the filter receives an HTML fragment, it delegates the processing to the HTML filter. For example, if the JSON processor receives the following input:

```
{"p1", "foo<script>alert('xss')</script>"}
```

Then the JSON parser passes the JSON value `"...<script>alert('xss')</script>"` to the JSON filter. However the value is passed directly to the HTML filter since it is an HTML fragment. After the HTML filtering, the JSON processor returns the following result:

```
{"p1", "foo"}
```

When the JSON filter receives a JSON value that is not an HTML fragment, it handles the value as an attribute value in HTML documents. If the value includes any malicious keyword such as `"javascript:"`, it returns an empty string.

If the JSON processor receives a JSON value that is not a JSON-formatted string, then it invokes the JSON filter directly. For example, when the processor receives the following input,

```
"foojavascript:alert('xss')"
```

then it invokes the JSON filter to filter the JSON value. After that, it returns the empty string.

At this time, since there is no configuration for the JSON filtering, the JSON filtering must be invoked as described in the Steps to invoke the ACF section

6.3 RSS/ATOM feed filtering

Recently, some reports alert that there is a XSS vulnerability in some RSS/ATOM feed readers. If the readers receive the RSS/ATOM feed which includes a malicious script, they might run the script unconciously. In order to prevent such new kind of XSS attack, ACF provides a RSS/ATOM feed filter for RSS/ATOM feed filtering.

6.3.1 RSS/ATOM feed filter

Suppose the following RSS/ATOM feed.

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>id_1</id>
  <title type="text">title_1</title>
  <entry xmlns="http://www.w3.org/2005/Atom">
    <title type="text">title_1_1</title>
    <id>id_1_1</id>
    <content type="text/html">
      <![CDATA[<script>alert('xss');</script>]]>
    c
```

When the RSS/ATOM feed processor receives the RSS/ATOM feed above, it invokes a XML parser to parse the feed. While parsing the feed, the XML parser passes any characters and any CDATA section to the RSS/ATOM feed filter. When the RSS/ATOM feed filter receives an HTML fragment, it delegates the processing to the HTML filter. In case of the feed above, it outputs the following feed after filtering process.

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>id_1</id>
  <title type="text">title_1</title>
  <entry xmlns="http://www.w3.org/2005/Atom">
    <title type="text">title_1_1</title>
    <id>id_1_1</id>
    <content type="text/html">
      <![CDATA[]]>
    c
```

When the RSS/ATOM feed filter receives a non-HTML fragment, it handles the value as an attribute value in HTML documents. If the value includes any malicious keyword such as `"javascript:"`, it returns an empty string.

At this time, since there is no configuration for the RSS/ATOM feed filtering, the RSS/ATOM feed filtering must be invoked as described in the Steps to invoke the ACF section

7. Legacy support

7.1 XPath-based HTML filtering

7.1.1 Invokeing XPath-based HTML filtering

The following methods are available in the ACF factory (`com.ibm.trl.acf.api.ActiveContentProcessorFactory`) to call an ACF processor (`com.ibm.trl.acf.api.ActiveContentProcessor`) and invoke the ACF. For more details, please see the Invoking the ACF section.

- **`getActiveContentProcessor(String contentType, String httpUserAgent, String ruleSetName)`** (deprecated): Get the ACF processor associated with the content type, the HTTP user agent, and the name of a rule set. Since the HTTP user agent is ignored in the current implementation, the following rule set names identify the configuration file. This method is retained only for backward compatibility with the previous version of the ACF.
  - **`"default"`** or **`"for-fragments"`**: This uses the `"acf-xpathconfig.xml"` configuration file.
  - **`"silent"`**: This uses the `"acf-xpathconfig-wComment.xml"` configuration file.

One of the following methods is available in the ACF processor to filter input HTML document and get the filtered document.

- **`boolean process(java.io.Reader source, java.io.Writer sink)`** **(deprecated)**: The ACF processor reads input from the specified source (reader or input stream) and filters the input. The result of filtering is written into the specified sink (writer or output stream). This method returns true if there is any malicious content in the input.

One of the following methods is available in the ACF processor to validate if there is any malicious content check if there is any filter input HTML document in the input.

- **`void validate(java.io.Reader source, java.io.Writer sink)`** **(deprecated)**: The ACF processor reads input from the specified source (reader or input stream) and validates the input. The result of filtering is written into the specified sink (writer or output stream) if there is no malicious content in the input. However, if there is any malicious content in the input, it throws an exception.

- **`String validate(java.io.Reader source)`** **(deprecated)**: These three validate method can be used. The difference between these and the methods above is what the ACF processor does when there is any malicious content in the input. In this case, this method returns the malicious content itself without throwing any exception.

Here are a code sample invoking XPath-based HTML filtering.

Example 1: Invoke XPath-based HTML filtering. (deprecated)

```
import com.ibm.trl.acf.api.ActiveContentProcessor;
import com.ibm.trl.acf.api.ActiveContentProcessorFactory;
import com.ibm.trl.acf.lookup.ActiveContentProcessorFactoryHome;

// 1. Get an ActiveContentProcessorFactory instance.
ActiveContentProcessorFactory factory =
ActiveContentProcessorFactoryHome.getActiveContentProcessorFacory();

// 2. Get an ActiveContentProcessor instance.
ActiveContentProcessor processor = factory.getActiveContentProcessor("text/html");

// 3. Invoke the HTML filtering.
Reader source = ...; // Includes an HTML document or an HTML fragment
Writer sink = ...; // Used to output the result of HTML filtering
processor.process(source, sink);
```

7.1.2 Configuration for XPath-based HTML filtering

The ACF is configured in a declarative way using a configuration file written in XML. ACF provides the following default configuration files for the HTML filtering in the `com.ibm.trl.acf.impl.html` package. These files for XPath-based HTML filtering are loaded during the startup.

1. **`acf-xpathconfig.xml`** (deprecated): ACF is configured with the HTML base filter only. The filter rules defined in this file are the same as in the `acf-annotationconfig.xml` file. The difference between two is that this file uses XPath to limit the scope of the HTML filtering.
2. **`acf-xpathconfig-wComment.xml`** (deprecated): ACF is configured with the HTML base filter only. In the HTML

base filter, all elements and attribute values that may include malicious active content are removed. When an element or an attribute value is removed, a comment indicating what tag or attribute value was removed is added at that location.

3. **`acf-xpathconfig-flash.xml`** (deprecated): ACF is configured with the HTML base filter and Flash filter. The filter rules defined in this file are the same as in the `acf-annotationconfig-flash.xml` file. The difference between the two is that this file uses XPath to limit the scope of the HTML filtering.

This section describes general parts of the ACF configuration file.

7.1.3 How to write a configuration file for XPath-based HTML filtering

A configuration file is composed of two main parts. In the first part, a chain of filters is declared. In the second part, the detailed filter rules are declared for the filters that were specified in the first part(See Figure 12).

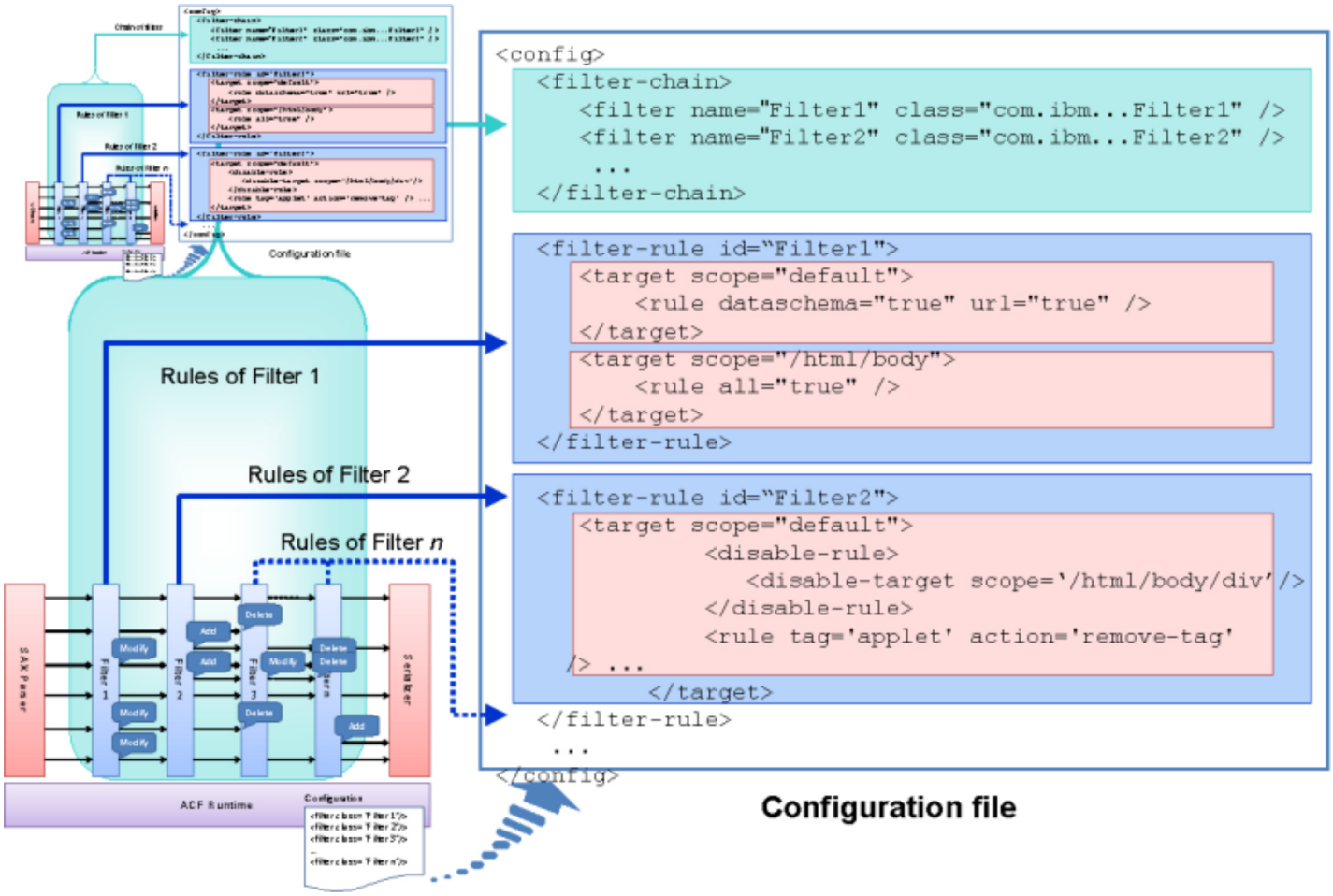

Figure 12: Composition of the configuration file

Here is the detail of the elements in the configuration file.

- **`config`**: This element is the root element of the ACF configuration file. It can be include only one filter chain.
- **`filterChain`**: This element contains a chain of filters. Each filter, described as a child element of this element, is applied in the order of appearance.
- **`filter`**: This element contains the basic information for a filter. This element can appear 0 (zero) or more times in the configuration file. The attributes of this element are:
  - **`@name`** (Required): An arbitrary name for the filter.
  - **`@class`** (Required): A name for the class that extends the `com.ibm.trl.acf.api.Filter` abstract class.
  - **`@verbose-output`** (Optional): The default value is `false`. If it is set to `true`, a comment is inserted whenever active content is removed.
  - **`@use-annotation`** (Optional): The default value is `true`. This attribute is used only for the HTML filtering. If it

is set to `false`, XPath is used to limit the scope of the HTML filtering.

- **`filter-rule`**: The actual filtering rule for each filter declared in the <filter> elements in the <filter-chain>. Each filter chain is required to have at least one filter rule. The attribute of this element is:
  - **`@id`** (Required): The associated identifier of the filter rule that is specified in the <filter> as `filter/@name`.
- **`target`**: This element specifies the filtering scope, and may appear one or more times. Here is the attribute of this element:
  - **`@scope`** (Required): This attribute is an XPath expression used to limit the filtering scope. When the value is empty or "default", the rules are applied to all of the input.

Here is an example of a `target` element:

```
<target scope="/html/body">
```

- **`rule`**: This element contains the parameters of the filter as custom attributes. The attributes that are defined depend upon the filter being configured.
- **`disable-rule`**: This element has child elements that specify scopes where the filter is not applied. Specified elements can be excluded from the scope specified by using the `disable-target/@scope` for the target of the HTML filtering.
- **`disable-target`**: This element has child elements that specify scopes where the filter is not applied. The attribute of this element is:
  - **`@scope`** (Required): The value is described using the XPath expression such as `target/@scope`.

For example, suppose the following configuration file for XPath-based HTML filtering. However, **because ofa vulnerability in XPath-based HTML filtering (as described in the annotation-based HTML filtering section), we recommend using annotation-based HTML filtering whenever possible.**

```
<config>
   <filter-chain>
      <filter name="Filter" class="...">
   </filter-chain>
   <filter-rule id="Filter">
      <target scope="/html/head">
         <rule .../>
         <rule .../>
          ...
      </target>
      <target scope="/html/body">
         <rule .../>
          ...
         <disable-rule>
            <disable-target scope="/html/body/iframe">
            <disable-target scope="/html/body/div">
         </disable-rule>
      </target>
   </filter-rule>
</config>
```

In this case, the ACF filters the input HTML document as shown in Figure 13.

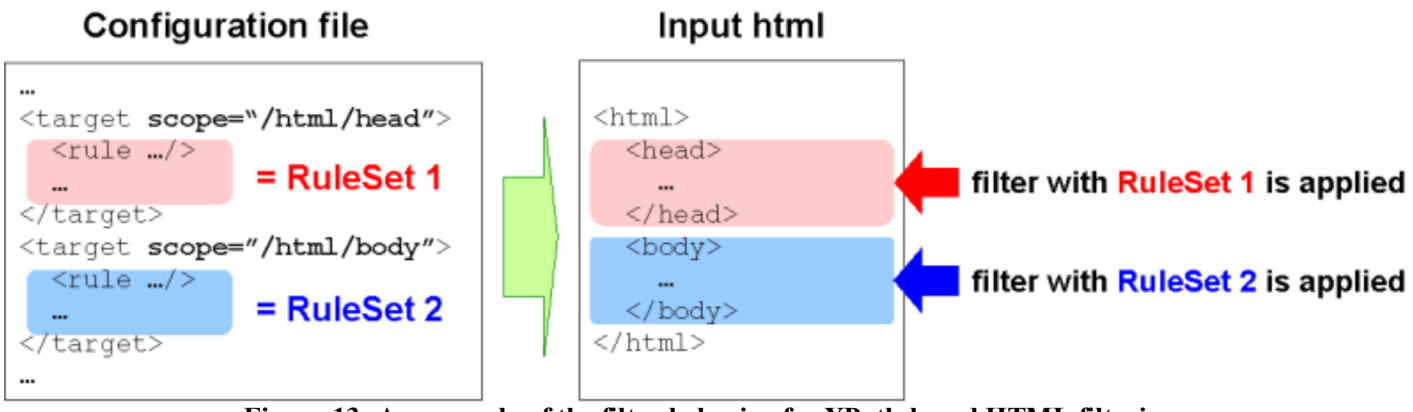

Figure 13: An example of the filter behavior for XPath-based HTML filtering

7.1.4 Limited XPath for scope

The XPath expressions that you can use in ACF are limited. They can contain only limited axis and expressions, so "/", "//", "[]", and "@=". For example, "/html/body" is valid but "//*[count(*)=3]" is not. In this example, the last (forbidden) expression would select all elements with 3 children and is valid according to the XPath specification.

7.1.5 HTML base filter for XPath-based HTML filtering

XPath-based HTML filtering can be used when `"config/filter-chain/filter@use-annotation"` is set to `"false"` for the HTML base filter The scope of the HTML filtering using XPath can be limited as shown in the Limited XPath for scope section to improve the performance of the HTML filtering. For example, if the ACF receives the following HTML document and the XPath `"/html/body//div"` is specified in the configuration,

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>
            <!-- user input here -->
            hello<br>
            <script>alert('xss')</script>
          </div>
        </td>
      </tr>
    <table>
  </body>
</html>
```

then the results of the filtering appear below, where the filtering scope is shown in red.

```
<html>
  <body>
    <table>
      <tr>
        <td>
          <div>
            <!-- user input here -->
            hello<br>

          </div>
        </td>
      </tr>
    <table>
  </body>
</html>
```

Configuration

Please see the configuration for annotation-based HTML filtering section for details of the configuration. Here are sample configuration files for XPath-based HTML filtering.

Example 1: The HTML base filter with an extra rule that removes script tags in the input is applied.

```
<config>
   <filter-chain>
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter"
use-annotation="false" />
   </filter-chain>

   <filter-rule id="base">
      <target scope="">
         <rule c14n="true" all="true" />
```

```
         <rule tag="script" action="remove-tag" />
      </target>
   </filter-rule>
</config>
```

The results of this example configuration appear in Figure 14.

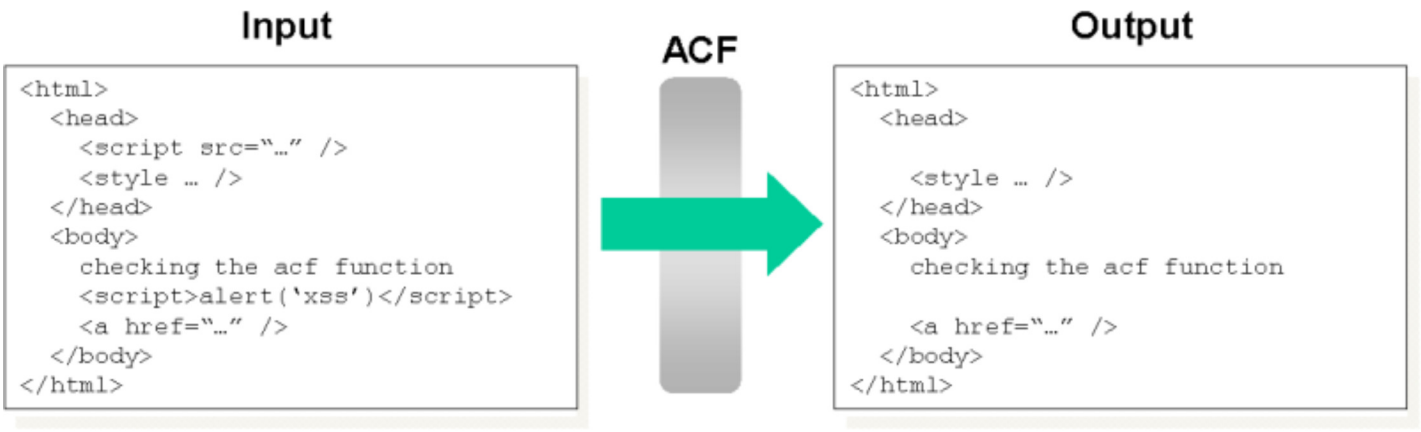

Figure 14: Input and output when the configuration for XPath-based HTML filtering is used with the filter rule

Example 2: The HTML base filter with two rules that remove scripts and style tags, where the rules are applied only inside the scope defined by the XPath expression `"/html/head"`

```
<config>
   <filter-chain>
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter"
use-annotation="false" />
   </filter-chain>

   <filter-rule id="base">
      <target scope="/html/head">
         <rule c14n="true" all="true" />
         <rule tag="script" action="remove-tag" />
         <rule tag="style" action="remove-tag" />
      </target>
   </filter-rule>
</config>
```

Example 3: Both the attribute value canonicalizer and the HTML base filters are used. If there are encoded strings in the input, they are decoded by the attribute value canonicalizer.

```
<config>
   <filter-chain>
      <filter name="c14n" class="com.ibm.trl.acf.impl.html.canonicalizer.C14NFilter"
use-annotation="false" />
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter"
use-annotation="false" />
   </filter-chain>

   <filter-rule id="c14n">
      <target scope="">
         <rule all="true" />
      </target>
   </filter-rule>

   <filter-rule id="base" >
      <target scope="">
         <rule attribute="href" value="javascript" value-criterion="starts-with"
action="remove-attribute-value" />
         <rule attribute="src" value="javascript" value-criterion="starts-with"
action="remove-attribute-value" />
```

```
      </target>
   </filter-rule>
</config>
```

Example 4: The attribute value canonicalizer and the base filter are used. Multiple scopes and the `disable-rule` scope are used.

```
<config>
   <filter-chain>
      <filter name="c14n" class="com.ibm.trl.acf.impl.html.canonicalizer.C14NFilter"
use-annotation="false" />
      <filter name="base" class="com.ibm.trl.acf.impl.html.basefilter.BaseFilter"
use-annotation="false" />
   </filter-chain>

   <filter-rule id="c14n">
      <target scope="">
         <rule all="true" />
      </target>
   </filter-rule>

   <filter-rule id="base" >
      <target scope="/html/body/div">
         <rule tag="script" action="remove-tag" />
         <disable-rule>
            <disable-target scope="html/body/div[@class="safe"]">
         </disable-rule>
      </target>

      <target scope="/html/head">
         <rule attribute="style" value="expression" value-criterion="contains"
action="remove-attribute-value" />
      </target>
   </filter-rule>
</config>
```