DOM based XSS Prevention Cheat Sheet

From OWASP

Contents

Introduction

When looking at XSS (Cross Site Scripting) there are three generally recognized forms of XSS. Reflected, Persisted, and DOM based XSS. The XSS Prevention Cheatsheet (http://www.owasp.org/index.php /XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet) has done an excellent job of addressing Reflected and Persisted XSS. This cheatsheet will address DOM (Document Object Model) based XSS and is an extension (as assumes comprehension of) the the XSS Prevention Cheatsheet (http://www.owasp.org/index.php /XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet) .

In order to understand DOM based XSS one needs to see the fundamental difference

between reflected and persisted XSS when compared to DOM based XSS. Reflected and persisted XSS exist in a higher level rendering context and DOM based XSS is primarily found in a lower level execution context. A rendering context is associated with the parsing of HTML tags and their attributes. The HTML parser of the rendering context dictates how data is presented and laid out on the page and can be further broken down into the standard contexts of HTML, HTML attribute, URL, and CSS. The JavaScript or VBScript parser of an execution context is associated with the parsing and execution of script code. Each parser has distinct and separate semantics in the way they can possibly execute script code (XSS) which make creating consistent rules for mitigating both rendering and execution based contexts difficult. The complication is compounded by the differing meanings and treatment of encoded values within each sub context (HTML, HTML attribute, URL, and CSS) within the execution context.

This paper refers to the HTML, HTML attribute, URL, and CSS Cheatsheet contexts as subcontexts because each of these contexts can be reached and set within a JavaScript execution context. In JavaScript code the main context is JavaScript but since an attacker can try to attack the other 4 contexts using equivalent JavaScript DOM methods we refer to the other contexts besides the JavaScript context as sub contexts.

The following example of an attack which occurs in the JavaScript context and HTML sub context:

```
<script>
var x = '<%= htmlAndJavaScriptEncodedVar %>';
var d = document.createElement('div');
d.innerHTML = x;
document.body.appendChild(d);
</script>
```

One consistency, however, is the need to JavaScript encode in addition to the encoding required for the sub context in the execution context. Let's look at the individual sub contexts of the execution context in turn.

HTML Sub Context within the Execution Context

There are several methods and attributes which can be used to directly render HTML content within JavaScript. These methods constitute the HTML Sub Context within the Execution Context.

Attributes

```
element.innerHTML = “<HTML> Tags and markup”;
element.outerHTML = “<HTML> Tags and markup”;
```

Methods

```
document.write(“<HTML> Tags and markup”);
document.writeln(“<HTML> Tags and markup”);
```

Guideline

In a pure HTML execution context (not HTML Attribute) use HTML and JavaScript encoding to mitigate against attacks.

```
element.innerHTML = “<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”;
element.outerHTML = “<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”;
```

Methods

```
document.write(“<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”);
document.writeln(“<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>”);
```

HTML Attribute Sub Context within the Execution Context

The HTML attribute Sub Context within the Execution context is divergent from the standard encoding rules. This is because the rule to HTML attribute encode in an HTML attribute rendering context is mitigating attacks which try to exit out of the attribute to add additional attributes and/or tags which could have executable code. When you are in a DOM execution context you only need to JavaScript encode HTML attributes which do not execute code (attributes other than event handler, CSS, and URL attributes).

For example, the general rule is to HTML Attribute encode untrusted data (data from the database, http request, user, backend system, etc.) placed in an HTML Attribute. This is the appropriate step to take when outputting data in a rendering context, however using HTML Attribute encoding in an execution context will break the application display of data.

```
var x = document.createElement(“input”);
x.setAttribute(“name”, “company_name”);
x.setAttribute(“value”, ‘<%=Encoder.encodeForJS(Encoder.encodeForHTMLAttr(companyName))%>’);
var form1 = document.forms[0];
form1.appendChild(x);
```

The problem is that if companyName had the value “Johnson & Johnson”. What would be displayed in the input text field would be “Johnson & Johnson”. The appropriate encoding to use in the above case would be only JavaScript encoding to disallow an attacker from closing out the single quotes and in-lining code, or escaping to HTML and opening a new script tag.

```
var x = document.createElement(“input”);
x.setAttribute(“name”, “company_name”);
x.setAttribute(“value”, ‘<%=Encoder.encodeForJS(companyName)%>’);
var form1 = document.forms[0];
form1.appendChild(x);
```

It is important to note that when setting an HTML attribute which does not execute code the value is set directly within the object attribute of the HTML element so there is no concerns with injecting up.

URL Attribute Sub Context within the Execution Context

The logic which parses URLs in both execution and rendering contexts looks to be the same. Therefore there is little change in the encoding rules for URL attributes in an execution (DOM) context.

```
var x = document.createElement(“a”);
x.setAttribute(“href”, ‘<%=Encoder.encodeForJS(Encoder.encodeForURL(userRelativePath))%>’);
var y = document.createTextElement(“Click Me To Test”);
x.appendChild(y);
document.body.appendChild(x);
```

If you utilize fully qualified URLs then this will break the links as the colon in the protocol identifier (“http:” or “javascript:”) will be URL encoded preventing the “http” and “javascript” protocols from being invoked.

CSS Attribute Sub Context within the Execution Context

Normally executing JavaScript from a CSS context required either passing `javascript:attackCode()` to the CSS url() method or invoking the CSS expression() method passing JavaScript code to be directly executed. From my experience, calling the

expression() function from an execution context (JavaScript) has been disabled. In order to mitigate against the CSS url() method ensure that you are URL encoding the data passed to the CSS url() method.

```
document.body.style.backgroundImage = "url(<%=Encoder.encodeForJS(Encoder.encodeForURL(companyName))%>)";
```

TODO: We have not been able to get the expression() function working from DOM JavaScript code. Need some help.

Event Handler and JavaScript code Sub Contexts within an Execution Context

Putting dynamic data within JavaScript code is especially dangerous because JavaScript encoding has different semantics for JavaScript encoded data when compared to other encodings. In many cases, JavaScript encoding does not stop attacks within an execution context. For example, a JavaScript encoded string will execute even though it is JavaScript encoded.

```
var x = document.createElement("a");
x.href="#”;
x.setAttribute("onclick", "\u0061\u006c\u0065\u0072\u0074\u0028\u0032\u0032\u0029");
var y = document.createTextNode("Click To Test");
x.appendChild(y);
document.body.appendChild(x);
```

JavaScript event handler methods are dangerous because they implicitly do an eval() on the data passed to the DOM attribute. There are other places in JavaScript where JavaScript encoding is accepted as valid executable code.

```
for ( var \u0062=0; \u0062 < 10; \u0062++){
    \u0064\u006f\u0063\u0075\u006d\u0065\u006e\u0074
    .\u0077\u0072\u0069\u0074\u0065\u006c\u006e
    ("\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064");
}
\u0077\u0069\u006e\u0064\u006f\u0077
.\u0065\u0076\u0061\u006c
\u0064\u006f\u0063\u0075\u006d\u0065\u006e\u0074
.\u0077\u0072\u0069\u0074\u0065(111111111));
```

or

```
var s = "\u0065\u0076\u0061\u006c";
var t = "\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0031\u0029";
window[s](t);
```

Because JavaScript is based on an international standard (ECMAScript), JavaScript encoding enables the support of international characters in programming constructs and variables in addition to alternate string representations (string escapes).

However the opposite is the case with HTML encoding. HTML tag elements are well defined and do not support alternate representations of the same tag. So HTML encoding cannot be used to allow the developer to have alternate representations of the <a> tag for example.

HTML Encoding's Disarming Nature

In general, HTML encoding serves to castrate HTML tags which are placed in HTML and HTML attribute contexts. Working example (no HTML encoding):

```
<a href="…" >
```

Normally encoded example (Does Not Work – DNW):

```
&#x3c;a href=… &#x3e;
```

HTML encoded example to highlight a fundamental difference with JavaScript encoded values (DNW):

```
<&#x61; href=…>
```

If HTML encoding followed the same semantics as JavaScript encoding. The line above could have possibily worked to render a link. This difference makes JavaScript encoding a less viable weapon in our fight against XSS.

Guidelines for Developing Secure Applications Utilizing JavaScript

DOM based XSS is extremely difficult to mitigate against because of its large attack surface and lack of standardization across browsers. The guidelines below are an attempt to provide guidelines for developers when developing Web based JavaScript applications (Web 2.0) such that they can avoid XSS.

1. Untrusted data should only be treated as displayable text. Never treat untrusted data as code or markup within JavaScript code.
2. Always JavaScript encode and delimit untrusted data as quoted strings when entering the application (Jim Manico and Robert Hansen)

```
var x = "<%=encodedJavaScriptData%>";
```

3. Use `document.createElement("…"), element.setAttribute("…","value"), element.appendChild(…),` etc. to build dynamic interfaces. Avoid use of HTML rendering methods:

1. `element.innerHTML = "…";`
2. `element.outerHTML = "…";`
3. `document.write(…);`
4. `document.writeln(…);`

4. Understand the dataflow of untrusted data through your JavaScript code. If you do have to use the methods above remember to HTML and them JavaScript encode the untrusted data (Stefano Di Paola).
5. There are numerous methods which implicitly eval() data passed to it. Make sure that any untrusted data passed to these methods is delimited with string delimiters and enclosed within a closure or JavaScript encoded to N-levels based on usage, and wrapped in a custom function. Ensure to follow step 4 above to make sure that the untrusted data is not sent to dangerous methods within the custom function or handle it by adding an extra layer of encoding.

Utilizing an Enclosure (as suggested by Gaz)

The example that follows illustrates using closures to avoid double JavaScript encoding.

```
setTimeout((function(param) { return function() {
         customFunction(param);
         }
})("<%=Encoder.encodeForJS(untrustedData)%>"), y);
```

The other alternative is using N-levels of encoding.

N-Levels of Encoding

If your code looked like the following, you would need to only double JavaScript encode input data.

```
setTimeout("customFunction('<%=doubleJavaScriptEncodedData%>', y)");
function customFunction (firstName, lastName)
      alert("Hello" + firstName + " " + lastNam);
}
```

The `doubleJavaScriptEncodedData` has its first layer of JavaScript encoding reversed in the single quotes. Then the implicit `eval()` of `setTimeout()` reverses another layer of JavaScript encoding to pass the correct value to `customFunction`. The reason why you only need to double JavaScript encode is that the `customFunction` function did not itself pass the input to another method which implicitly or explicitly called `eval()`. If "firstName" was passed to another JavaScript method which implicitly or explicitly called eval() then `<%=doubleJavaScriptEncodedData%>` above would need to be changed to `<%=tripleJavaScriptEncodedData%>`.

An important implementation note is that if the JavaScript code tries to utilize the double or triple encoded data in string comparisons, the value may be interpreted as different values based on the number of evals() the data has passed through before

being passed to the if comparison and the number of times the value was JavaScript encoded.

If "A" is double JavaScript encoded then the following if check will return false.

```
var x = "doubleJavaScriptEncodedA";  //\u005c\u0075\u0030\u0030\u0034\u0031
if (x == "A") {
   alert("x is A");
} else if (x == "\u0041") {
   alert("This is what pops");
}
```

This brings up an interesting design point. Ideally, the correct way to apply encoding and avoid the problem stated above is to server-side encode for the output context where data is introduced into the application. Then client-side encode (using a JavaScript encoding library such as ESAPI4JS) for the individual sub context (DOM methods) which untrusted data is passed to. ESAPI4JS (located at http://bit.ly/9hRTLH) and jQuery Encoder (located at https://github.com/chrisisbeef/jquery-encoder/blob/master/src/main/javascript/org/owasp/esapi/jquery/encoder.js) are two client side encoding libraries developed by Chris Schmidt.
Here are some examples of how they are used:

```
var input = “<%=Encoder.encodeForJS(untrustedData)%>”;  //server-side encoding
```

```
window.location = ESAPI4JS.encodeForURL(input);  //URL encoding is happening in JavaScript
```

```
document.writeln(ESAPI4JS.encodeForHTML(input));  //HTML encoding is happening in JavaScript
```

It has been well noted by the group that any kind of reliance on a JavaScript library for encoding would be problematic as the JavaScript library could be subverted by attackers. One option is to wait till ECMAScript 5 so the JavaScript library could support immutable properties.
Another option provided by Gaz (Gareth) was to use a specific code construct to limit mutability with anonymous clousures.

An example follows:

```
function escapeHTML(str) {
     str = str + "";
     var out = "";
     for(var i=0; i<str.length; i++) {
         if(str[i] === '<') {
             out += '&lt;';
         } else if(str[i] === '>') {
             out += '&gt;';
         } else if(str[i] === "'") {
             out += '&#39;';
         } else if(str[i] === '"') {
             out += '&quot;';
         } else {
             out += str[i];
         }
     }
     return out;
}
```

Chris Schmidt has put together another implementation of a JavaScript encoder at http://yet-another-dev.blogspot.com/2011/02/client-side-contextual-encoding-for.html.

6. Limit the usage of dynamic untrusted data to right side operations. And be aware of data which may be passed to the application which look like code (eg. `location`, `eval()`). (Achim)

```
var x = “<%=properly encoded data for flow%>”;
```

If you want to change different object attributes based on user input use a level of indirection.

Instead of:

```
window[userData] = “moreUserData”;
```

Do the following instead:

```
if (userData===”location”) {
   window.location = “static/path/or/properly/url/encoded/value”;
}
```

7. When URL encoding in DOM be aware of character set issues as the character set in JavaScript DOM is not clearly defined (Mike Samuel).

8. Limit access to properties objects when using object[x] accessors. (Mike Samuel). In other words use a level of indirection between untrusted input and specified object properties.
Here is an example of the problem when using map types:

```
var myMapType = {};
myMapType[<%=untrustedData%>] = “moreUntrustedData”;
```

Although the developer writing the code above was trying to add additional keyed elements to the `myMapType` object. This could be used by an attacker to subvert internal and external attributes of the `myMapType` object.

9. Run your JavaScript in a ECMAScript 5 canopy or sand box to make it harder for your JavaScript API to be compromised (Gareth Heyes and John Stevens).

10. Don’t `eval()` JSON to convert it to native JavaScript objects. Instead use `JSON.toJSON()` and `JSON.parse()` (Chris Schmidt).

Common Problems Associated with Mitgating DOM Based XSS

Complex Contexts

In many cases the context isn’t always strait forward to discern.

```
<a href=”javascript:myFunction(‘<%=untrustedData%>’, 'test');”>Click Me</a>
...
<script>
Function myFunction (url,name) {
    window.location = url;
}
</script>
```

In the above example, untrusted data started in the rendering URL context (`href` attribute of an `<a>` tag) then changed to a JavaScript execution context (`javascript:` protocol handler) which passed the untrusted data to an execution URL sub context (`window.location` of myFunction). Because the data was introduced in JavaScript code and passed to a URL sub context the appropriate server-side encoding would be the following:

```
<a href=”javascript:myFunction(‘<%=Encoder.encodeForJS( ↩
              Encoder.encodeForURL(untrustedData))%>’, 'test');”>Click Me</a>
…
```

Or if you were using ECMAScript 5 with an immutable JavaScript client-side encoding libraries you could do the following:

```
<!--server side URL encoding has been removed.  Now only JavaScript encoding on server side. -->
<a href="javascript:myFunction('<%=Encoder.encodeForJS(untrustedData)%>', 'test');">Click Me</a>
...
<script>
Function myFunction (url,name) {
    var encodedURL = ESAPI4JS.encodeForURL(url);  //URL encoding using client-side scripts
    window.location = encodedURL;
}
</script>
```

Insonsistencies of Encoding Libraries

There are a number of open source encoding libraries out there:

1. ESAPI
2. Apache Commons String Utils
3. Jtidy
4. Your company's custom implementation.

Some work on a black list others ignore important characters like "<" and ">". ESAPI is one of the few which work on a whitelist and encode all non-alpha numeric characters. It is important to use an encoding library which understands which characters can be used to exploit vulnerabilies in their respective contexts. But there are misconceptions abound related to proper encoding.

Encoding Misconceptions

Many security training curriculums and papers advocate the blind usage of HTML encoding to resolve XSS. This logically seems to be prudent advice as the JavaScript parser does not understand HTML encoding. However, if the pages returned from your web application utilize a content type of "text/xhtml" or the file type extension of "*.xhtml" then HML encoding may not work to mitigate against XSS.

For example:

```
<script>
&#x61;lert(1);
</script>
```

The HTML encoded value above is still executable. If that isn't enough to keep in mind, you have to remember that encodings are lost when you retrieve them using the value attribute of a DOM element.

Let's look at the sample page and script:

```
<form name=”myForm” …>
  <input type=”text” name=”lName” value=”<%=Encoder.encodeForHTML(last_name)%>”>
…
</form>
<script>
var x = document.myForm.lName.value;  //when the value is retrived the encoding is reversed
document.writeln(x);  //any code passed into lName is now executable.
</script>
```

Finally there is the problem that certain methods in JavaScript which are usually safe can be unsafe in certain contexts.

Usually Safe Methods

One example of an attribute which is usually safe is innerText. Some papers or guides advocate its use as an alternative to innerHTML to mitigate against XSS in innerHTML. However, depending on the tag which innerText is applied, code can be executed.

```
<script>
var tag = document.createElement(“script”);
tag.innerText = “<%=untrustedData%>”;  //executes code
</script>
```

Authors and Contributing Editors

Jim Manico - jim[at]owasp.org

Abraham Kang - abraham.kang[at]owasp.org

Gareth (Gaz) Heyes

Stefano Di Paola

Achim Hoffmann achim_@_owasp.org (mailto:achim@owasp.org|)

Robert (RSnake) Hansen

Mario Heiderich

John Stevens

Chris (Chris BEF) Schmidt

Mike Samuel

Jeremy Long

Edwardo (SirDarkCat) Alberto Vela Nava

Jeff Williams - jeff.williams[at]owasp.org

Erlend Oftedal

Other Articles in the OWASP Prevention Cheat Sheet Series

- Authentication Cheat Sheet
- Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
- Forgot Password Cheat Sheet
- Cryptographic Storage Cheat Sheet
- SQL Injection Prevention Cheat Sheet
- Transport Layer Protection Cheat Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet
- **DOM based XSS Prevention Cheat Sheet**

Categories: How To | Cheatsheets