DOM based XSS Prevention Cheat Sheet

Last revision (mm/dd/yy): **11/25/2017**

[hide]

Home
About OWASP
Acknowledgements
Advertising
AppSec Events
Books
Brand Resources
Chapters
Donate to OWASP
Downloads
Funding
Governance
Initiatives
Mailing Lists
Membership
Merchandise
News
Community portal
Presentations
Press
Projects
Video
Volunteer

Reference
Activities
Attacks
Code Snippets
Controls
Glossary
How To...
Java Project
.NET Project
Principles
Technologies
Threat Agents
Vulnerabilities

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

Introduction

When looking at XSS (Cross-Site Scripting), there are three generally recognized forms of XSS. Reflected, Stored, and DOM Based XSS. The XSS Prevention Cheatsheet does an excellent job of addressing Reflected and Stored XSS. This cheatsheet addresses DOM (Document Object Model) based XSS and is an extension (and assumes comprehension of) the XSS Prevention Cheatsheet.

In order to understand DOM based XSS, one needs to see the fundamental difference between Reflected and Stored XSS when compared to DOM based XSS. The primary difference is where the attack is injected into the application. Reflected and Stored XSS are server side injection issues while DOM based XSS is a client (browser) side injection issue. All of this code originates on the server, which means it is the application owner's responsibility to make it safe from XSS, regardless of the type of XSS flaw it is. Also, XSS attacks always **execute** in the browser. The difference between Reflected/Stored XSS is where the attack is added or injected into the application. With Reflected/Stored the attack is injected into the application during server-side processing of requests where untrusted input is dynamically added to HTML. For DOM XSS, the attack is injected into the application during runtime in the client directly.

When a browser is rendering HTML and any other associated content like CSS, JavaScript, etc. it identifies various rendering contexts for the different kinds of input and follows different rules for each 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 which make creating consistent rules for mitigating vulnerabilities in various contexts difficult. The complication is compounded by the differing meanings and treatment of encoded values within each subcontext (HTML, HTML attribute, URL, and CSS) within the execution context.

For the purposes of this article, we refer to the HTML, HTML attribute, URL, and CSS 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 with the right tags and context closing characters, an attacker can try to attack the other 4 contexts using equivalent JavaScript DOM methods.

The following is an example vulnerability which occurs in the JavaScript context and HTML subcontext:

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

Let's look at the individual subcontexts of the execution context in turn.

RULE #1 - HTML Escape then JavaScript Escape Before Inserting Untrusted Data into HTML Subcontext 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 Subcontext within the Execution Context. If these methods are provided with untrusted input, then an XSS vulnerability could result. For example:

Example Dangerous HTML Methods

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

To make dynamic updates to HTML in the DOM safe, we recommend a) HTML encoding, and then b) JavaScript encoding all untrusted input, as shown in these examples:

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

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

Note: The Encoder.encodeForHTML() and Encoder.encodeForJS() are just notional encoders. Various options for actual encoders are listed later in this document.

RULE #2 - JavaScript Escape Before Inserting Untrusted Data into HTML Attribute Subcontext within the Execution Context

The HTML attribute *subcontext* 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 necessary in order to mitigate attacks which try to exit out of an HTML attributes or try to add additional attributes which could lead to XSS. 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, back-end 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.

SAFE but BROKEN example

```
var x = document.createElement("input"); x.setAttribute("name",
"company_name"); // In the following line of code, companyName represents
untrusted user input // The Encoder.encodeForHTMLAttr() is unnecessary and
causes double-encoding 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.

SAFE and FUNCTIONALLY CORRECT example

```
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.

RULE #3 - Be Careful when Inserting Untrusted Data into the Event Handler and JavaScript code Subcontexts 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.

Therefore, the primary recommendation is to avoid including untrusted data in this context. If you must, the following examples describe some approaches that do and do not work.

```
var x = document.createElement("a"); x.href="#"; // In the line of code below,
the encoded data on the right (the second argument to setAttribute) // is an
example of untrusted data that was properly JavaScript encoded but still
executes. 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);
```

The setAttribute(*name_string*,*value_string*) method is dangerous because it implicitly coerces the *value_string* into the DOM attribute datatype of *name_string*. In the case above, the attribute name is an JavaScript event handler, so the attribute value is implicitly converted to JavaScript code and evaluated. In the case above, JavaScript encoding does not mitigate against DOM based XSS. Other JavaScript methods which take code as a string types will have a similar problem as outline above (setTimeout, setInterval, new Function, etc.). This is in stark contrast to JavaScript encoding in the event handler attribute of a HTML tag (HTML parser) where JavaScript encoding mitigates against XSS.

```
<!-- Does NOT work --> <a id="bb" href="#"
onclick="\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029"> Test Me</a>
```

An alternative to using Element.setAttribute(...) to set DOM attributes is to set the attribute directly. Directly setting event handler attributes will allow JavaScript encoding to mitigate against DOM based XSS. Please note, it is always dangerous design to put untrusted data directly into a command execution context.

```
<a id="bb" href="#"> Test Me</a>
```

```
//The following does NOT work because the event handler is being set to a
string. "alert(7)" is JavaScript encoded. document.getElementById("bb").onclick
= "\u0061\u006c\u0065\u0072\u0074\u0028\u0037\u0029"; //The following does NOT
work because the event handler is being set to a string.
document.getElementById("bb").onmouseover = "testIt"; //The following does NOT
work because of the encoded "(" and ")". "alert(77)" is JavaScript encoded.
document.getElementById("bb").onmouseover =
\u0061\u006c\u0065\u0072\u0074\u0028\u0037\u0037\u0029; //The following does
NOT work because of the encoded ";". "testIt;testIt" is JavaScript encoded.
document.getElementById("bb").onmouseover =
\u0074\u0065\u0073\u0074\u0049\u0074\u003b\u0074\u0065\u0073\u0074\u0049\u0074;
//The following DOES WORK because the encoded value is a valid variable name or
function reference. "testIt" is JavaScript encoded
document.getElementById("bb").onmouseover =
\u0074\u0065\u0073\u0074\u0049\u0074; function testIt() { alert("I was
called."); }
```

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):

```
&amp;#x3c;a href=... &amp;#x3e;
```

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

```
<&amp;#x61; href=...>
```

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

RULE #4 - JavaScript Escape Before Inserting Untrusted Data into the CSS Attribute Subcontext 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))%>)";
```

RULE #5 - URL Escape then JavaScript Escape Before Inserting Untrusted Data into URL Attribute Subcontext 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.

RULE #6 - Populate the DOM using safe JavaScript functions or properties

The most fundamental safe way to populate the DOM with untrusted data is to use the safe assignment property, textContent. Here is an example of safe usage.

```
<script> element.textContent = untrustedData; //does not execute code </script>
```

RULE #7 - Fixing DOM Cross-site Scripting Vulnerabilities

The best way to fix DOM based cross-site scripting is to use the right output method (sink). For example if you want to use user input to write in a *div tag* element don’t use innerHtml, instead use innerText/textContent. This will solve the problem, and it is the right way to re-mediate DOM based XSS vulnerabilities.

It is always a bad idea to use a user-controlled input in dangerous sources such as eval. 99% of the time it is an indication of bad or lazy programming practice, so simply don’t do it instead of trying to sanitize the input.

Finally, to fix the problem in our initial code, instead of trying to encode the output correctly which is a

hassle and can easily go wrong we would simply use element.textContent to write it in a content like this:

```
<b>Current URL:</b> <span id="contentholder"></span> ..... <script>
document.getElementById("contentholder").textContent = document.baseURI;
</script>
```

It does the same thing but this time it is not vulnerable to DOM based cross-site scripting vulnerabilities.

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.

GUIDELINE #1 - Untrusted data should only be treated as displayable text

Avoid treating untrusted data as code or markup within JavaScript code.

GUIDELINE #2 - Always JavaScript encode and delimit untrusted data as quoted strings when entering the application when building templated Javascript

Always JavaScript encode and delimit untrusted data as quoted strings when entering the application al illustrated in the following example.

```
var x = "<%= Encode.forJavaScript(untrustedData) %>";
```

GUIDELINE #3 - Use **document.createElement("..."), element.setAttribute("...","value"), element.appendChild(...)** and similar to build dynamic interfaces

document.createElement("..."), element.setAttribute("...","value"), element.appendChild(...) and similar are save ways to build dynamic interfaces.

Please note, **element.setAttribute** is only safe for a limited number of attributes. Dangerous attributes include any attribute that is a command execution context, such as onclick or onblur. Examples of safe attributes includes align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width.

GUIDELINE #4 - Avoid sending untrusted data into HTML rendering methods

Avoid populating the following methods with untrusted data.

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

GUIDELINE #5 - Avoid the numerous methods which implicitly eval() data passed to it

There are numerous methods which implicitly eval() data passed to it that must be avoided . 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 (upon execution) 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 subcontext (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
```

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

One option is utilize ECMAScript 5 immutable properties in the JavaScript library.
Another option provided by Gaz (Gareth) was to use a specific code construct to limit mutability with anonymous closures.

An example follows:

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

GUIDELINE #6 - Limit the usage of untrusted data to only right side

operations

Not only is it good design to limit the usage of untrusted data to right side operations, but also be aware of data which may be passed to the application which look like code (eg. **location**, **eval()**).

If you want to change different object attributes based on user input then 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"; }
```

GUIDELINE #7 - When URL encoding in DOM be aware of character set issues

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

GUIDELINE #8 - Limit access to properties objects when using object[x] accessors

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:

```
12var 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.

GUIDELINE #9 - Run your JavaScript in a ECMAScript 5 canopy or sandbox

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).

GUIDELINE #10 - Don't **eval()** JSON to convert it to native JavaScript objects

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

Common Problems Associated with Mitigating DOM Based XSS

Complex Contexts

In many cases the context isn't always straightforward 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 subcontext (**window.location** of myFunction). Because the data was introduced in JavaScript code and passed to a URL subcontext 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:

```
&lt;!--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>
```

Inconsistencies 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 while others ignore important characters like "<" and ">". ESAPI is one of the few which works on a whitelist and encodes all non-alphanumeric characters. It is important to use an encoding library that understands which characters can be used to exploit vulnerabilities in their respective contexts. Misconceptions abound related to the proper encoding that is required.

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 HTML encoding may not work to mitigate against XSS.

For example:

```
<script> &amp;#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 retrieved 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 thought to be 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. Also note, innerText is non standard and is not supported in FireFox

```
<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[at]owasp.org
Robert (RSnake) Hansen
Mario Heiderich
John Steven
Chris (Chris BEEF) Schmidt
Mike Samuel
Jeremy Long
Dhiraj Mishra - mishra.dhiraj[at]owasp.org
Eduardo (SirDarkCat) Alberto Vela Nava
Jeff Williams - jeff.williams[at]owasp.org
Erlend Oftedal

Other Cheatsheets

V - T - E	Cheat Sheets [Collapse]
Developer / Builder	3rd Party Javascript Management · Access Control · AJAX Security Cheat Sheet · Authentication (ES) · Bean Validation Cheat Sheet · Choosing and Using Security Questions · Clickjacking Defense · Credential Stuffing Prevention Cheat Sheet · Cross-Site Request Forgery (CSRF) Prevention · Cryptographic Storage · C-Based Toolchain Hardening · Deserialization · **DOM based XSS Prevention** · Forgot Password · HTML5 Security · HTTP Strict Transport Security · Injection Prevention Cheat Sheet · Injection Prevention Cheat Sheet in Java · JSON Web Token (JWT) Cheat Sheet for Java · Input Validation · Insecure Direct Object Reference Prevention · JAAS · Key Management · LDAP Injection Prevention · Logging · Mass Assignment Cheat Sheet · .NET Security · OS Command Injection Defense Cheat Sheet · OWASP Top Ten · Password Storage · Pinning · Query Parameterization · REST Security · Ruby on Rails · Session Management · SAML Security · SQL Injection Prevention · Transaction Authorization · Transport Layer Protection · Unvalidated Redirects and Forwards · User Privacy Protection · Web Service Security · XSS (Cross Site Scripting) Prevention · XML External Entity (XXE) Prevention Cheat Sheet
Assessment / Breaker	Attack Surface Analysis · REST Assessment · Web Application Security Testing · XML Security Cheat Sheet · XSS Filter Evasion
Mobile	Android Testing · IOS Developer · Mobile Jailbreaking
OpSec / Defender	Virtual Patching · Vulnerability Disclosure
Draft and Beta	Application Security Architecture · Business Logic Security · Content Security Policy · Denial of Service Cheat Sheet · Grails Secure Code Review · IOS Application Security Testing · PHP Security · Regular Expression Security Cheatsheet · Secure Coding · Secure SDLC · Threat Modeling
All Pages In This Category	

Category: Cheatsheets