WHAT EVERY DEVELOPER SHOULD KNOW ABOUT XSS

Table of Contents

Fixing XSS: A Practical Guide for Developers

This document exposes the most common remediations that a you need to use when developing a web application in order to fix cross-site scripting (XSS).

We will first give some background information about HTML contexts, describe the Coverity Security Library, and jump into 13 common locations where dynamic data can appear in a web page.

Top 3 Things to Know about XSS Mitigation

Cross-site scripting (XSS) is a complex problem with many moving parts, but we want to highlight the most important "gotchas." These are the Top 3!

HTML escaping isn't enough

It is important to understand that HTML escaping (using HTML entities) is not always the right solution to output dynamic data in an HTML page. There is no magic escaper that can make dynamic data safe for all possible HTML output contexts. You need to use an escaping or sanitization that depends on what part of the HTML this dynamic data is written to (e.g., URL, JavaScript string, etc.).

Your frameworks won't help much

Frameworks typically have escapers. However, unless you use a security-focused library like the Coverity® Security Library (CSL), you will not have access to all the escapers you need. For instance, you won't find a CSS string escaper in the Spring framework. Their escapers, such as HTML entities and JavaScript strings, are not designed for security and often don't consider all characters that could be important.

Nested contexts will blow your mind (inception style)

One of the most important concepts to understand to fix XSS is the nested HTML contexts. Here's an example that shows how to come up with the stack of nested contexts for a particular output in an HTML page, and determine what escaping needs to be performed.

Let's consider this HTML snippet:

```
<a href="javascript:hello('${content}')">...
```

In this example, `content` is the dynamic data that is written to the HTML page using the Java Expression Language (EL) notation (so, `${content}` is the interesting data).

Dissecting where `content` is added, we see that content is encapsulated inside a:

- HTML attribute: The value of the attribute is quoted using `"`
- URL: Since we're in an `href` attribute, we know this is supposed to be a URL for the browser
- JavaScript: We recognize the `javascript:` scheme and what comes after will be considered as JavaScript
- JavaScript string: `content` belongs in a string that is passed to the `hello()` JavaScript function

To understand the HTML contexts, we need to see what the browser will do to eventually get to this content in JavaScript:

1. HTML parser gets the content of the `href` attribute and unescape the HTML entities
2. Parse the resulting URL and recognize that this is a `javascript:` scheme
3. Take the content of the URL (after the `javascript:`) and URL decode it

4. Pass the content of the URL to JavaScript
5. The JavaScript parser runs and gets the JavaScript string that contains our content
6. The content of the JavaScript string is processed for string escape sequence: JavaScript string decoding

These steps indicate what decoding sequence the browser executes. You need to reproduce this in reverse order to make the content safe for its stack of HTML contexts:

1. Quoted HTML attribute
2. URL
3. JavaScript string

So if you want to safely output `content`, you need to do something like this:

```
<a href="javascript:hello('${cov:htmlEscape(cov:uriEncode(cov:jsStringEscape(content)))}')" >...
```

So how do you know which contexts need escaping? Do you need to have a full HTML5 parser in your head? Luckily, no. There are 13 key contexts that you really need to understand and account for almost all situations you're likely to run into. The rest of this paper goes into them.

If you need to inject dynamic data into a context not covered here, consult your local security expert. Don't try to figure it out by twiddling with your browser... this is stuff you really want to make sure is right.

Coverity Security Library

The Coverity Security Library (CSL) is a secure, small, convenient and free library that helps deal with dynamic data for HTML and SQL.

CSL is available on **GitHub** at https://github.com/coverity/coverity-security-library, and in **Maven Central**, at http://mvnrepository.com/artifact/com.coverity.security .

In this document, we use CSL as the reference implementation for escapers and sanitizers to help with inserting dynamic data into a web page. We also use the Java Expression Language (EL) notation. Note that plain Java functions are also available. Please refer to the documentation on GitHub to know which one to use.

Installation and Usage

Please refer to the documentation on GitHub and the code examples to see how to embed CSL in your application, and how to use it from Java, JSP or EL.

Getting It Right

HTML Normal Element

Scenario:
You want to display some dynamic data inside the body of a tag.

Getting it right:
This is the easiest case - just HTML escape the dynamic content:

```
<div>
  Hello ${cov:htmlEscape(name)}!
</div>
```

HTML Attribute

Scenario:
You want to dynamically construct an attribute value for a tag.

Getting it right:
First, ensure that you have quoted your attribute with either double or single quotes and not backticks (`). You should do this for all attributes. Never use unquoted HTML attributes!

Next, check if the attribute is a URL, JavaScript, CSS or other embedded language. If it is, find the section for that type of attribute.

If it is not one of those types of attributes, then use a HTML escaper:

```
<div data="${cov:htmlEscape(content)}">
```

JavaScript String Inside HTML Attribute

Scenario:
You want to insert some dynamic data into an attribute that contains JavaScript.

Getting it right:
First, ensure that the attribute is properly quoted, for more details see the HTML attribute section.

Next, ensure that you are trying to insert the data into a JavaScript string, and that the string is quoted. If you are trying to insert the data outside of a string, please see the full CSL documentation for an alternative to the JavaScript string escaper.

Next, recognize that this is a nested context and order is important. The order that the browser will decode this in is:

1. HTML Attribute
2. JavaScript String

So, you need to apply a JavaScript string escaper, and then apply a HTML escaper to your dynamic like this:

```
<div onclick="jsFunc('${cov:htmlEscape(cov:jsStringEscape(content))}')">
  Click me
</div>
```

Full URL Inside HTML Attribute

Scenario:
You want to insert a dynamic URL into an attribute that will understand the URL, such as a link or iframe.

Getting it right:
First, ensure that the attribute is properly quoted, for more details see the HTML attribute section.

Next, ensure that the URL is not something that needs to be restricted to a server you control, like an object tag for a flash file or a `script` tag.

Now, while this might seem like a nested context, full URLs cannot be escaped properly to stop XSS, however the CSL has a function that will transform potentially dangerous URLs such as those beginning with the `javascript:` scheme, to safe URLs called `asURL` that you can call as if it were an escaper. To use `asURL` correctly, you should apply it to a fully constructed URL.

So with that in mind, we apply our `asURL` function and an HTML escaper:

```
<a href="${cov:htmlEscape(cov:asURL(content))}">
  Click me
</a>
```

CSS String Inside HTML Attribute

Scenario:
You want to allow a user to control a CSS string such as a `font-family` attribute or CSS selector that you want to specify in an attribute.

Getting it right:
Hopefully you're getting the hang of this by now, so you're going to jump right in and:

1. Quote the attribute with `"`
2. Quote the CSS string `'`
3. Identify the context stack is CSS inside an attribute

Apply a CSS string escaper, then apply a HTML escaper:

```
<a style="font-family:'${cov:htmlEscape(cov:cssStringEscape(content))}'">
  Click me
</a>
```

JavaScript String

Scenario:
You want to pass some dynamic data to a JavaScript block by writing it into a string.

Getting it right:
By now this should seem like a nested context, but as anyone who has ever had to make a website work across multiple browsers should know, browsers love it when things are different and exciting and hard to get right!

Luckily, this isn't as hard as that, and all we need to do is apply a JavaScript string escaper. The tricky thing here is ensuring that your JavaScript string escaper escapes `<` as well as the typical string characters. If you're using CSL, we've already taken care of that for you. Here's an example of how to get it right:

```
<script type="text/javascript">
  var str = '${cov:jsStringEscape(content)}';
  // Do something with the string 'str'
</script>
```

HTML Inside JavaScript String

Scenario:
You want to insert dynamic data inside a piece of HTML that will be used from JavaScript later.

Getting it right:
Just like in the previous example, the dynamic data will be part of a JavaScript string, so you need to use a JavaScript string escaper. However, as the HTML content also needs to be escaped, we use an HTML escaper:

```
<div id="forMyContent"></div>
<script>
  var foo = "<h1>${cov:jsStringEscape(cov:htmlEscape(content))}</h1>";
  $("#forMyContent")
  .html(foo);
</script>
```

JavaScript Number

Scenario:
You want to pass a dynamic number to JavaScript by writing it into a script block.

Getting it right:
Since there is no sane way to convert arbitrary data to a number and preserve the meaning, we've provided a function in CSL that will ensure that the string you pass it is a number, or will output a default value of 0. You can use it safely in this type of scenario:

```
<script type="text/javascript">
  var num = ${cov:asNumber(content)};
  // Do something with the number 'num'
</script>
```

CSS String

Scenario:
You want to allow a user to control a CSS string such as a `font-family` directive, or a CSS selector that you want to specify in a `style` tag.

Getting it right:
Like JavaScript strings inside script blocks, this one is easy - just ensure your string is quoted and then use the CSS string escaper in CSL:

```
<style>
  #foo[id ~= '${cov:cssStringEscape(content)}'] {
    background-color: pink !important;
  }
</style>
```

URL Inside CSS String

Scenario:
You want to allow a user to control a CSS URL such as a background image, that you want to specify in a style tag.

Getting it right:
First, ensure that the CSS URL is quoted properly, using single or double quotes.

Next, we need to use a CSS string escaper. We highly recommend using the escaper in CSL and not just a regular JavaScript escaper, as the security requirements for CSS are slightly different from JavaScript strings.

Unlike URLs in JavaScript or HTML, you do not need to worry about the contents of the URL, since modern CSS parsers do not interpret `javascript:` or other special URL with harmful content (e.g., `data:` with HTML content, etc.).

```
<style>
  #foo {
    background: url('${cov:cssStringEscape(content)}');
  }
</style>
```

CSS Color

Scenario:
You want to allow a user to control a CSS color such as a background color.

Getting it right:
Since CSS colors are not specified inside strings, there is no escaping that can be done in this context and we must do some kind of validation or filtering. CSL provides a filter that will allow a color specified as hex or text, but will prevent injection attacks; it can be used in the following way:

```
<style>
  #foo {
    background-color: ${cov:asCssColor(content)};
  }
</style>
```

URL Fragment (hash) Inside HTML Attribute

Scenario:
You want to output a URL for which the fragment is dynamic.

Getting it right:
The fragment (after the `#` character) is just text for the URL. There is no escaping associated to it, so to output it, you need to use the parent context's escaper. It is sometimes URL encoded, but the browser won't decode it for you.

In the following example, we output the URL in an HTML `href` attribute and we ensure that the fragment is escaped for HTML.

```
<a href="/path/page?elmt=1#${cov:htmlEscape(content)}">
  Link name
</a>
```

URL Query String and Path Element

Scenario:
You want to output a URL that contains dynamic data in either the path or the query string. The other elements of the URL are not dynamic.

Getting it right:
These elements of the URL can be safely URL encoded: path, query string element name and query string element values. We use the CSL `uriEncode` function to do this. Note that this is not true for all URL encoders, and you might need to use the nested approach (using a HTML escaper and URL encoder) if you do not use CSL.

```
<!-- Assuming 'name' and 'value' are dynamic data -->
<a href="http://example.com/path/?${cov:uriEncode(qsElmtName)}=
  ${cov:uriEncode(qsElmtValue)}">
  Click me
</a>
```

About Coverity

Coverity, Inc., the leader in development testing, is the trusted standard for companies that need to protect their brands and bottom lines from software failures. More than 1,100 Coverity customers use Coverity's development testing platform to automatically test source code for software defects that could lead to product crashes, unexpected behavior, security breaches or catastrophic failure. Coverity is a privately held company headquartered in San Francisco. Coverity is funded by Foundation Capital and Benchmark Capital.

coverity®

For More Information
www.coverity.com
Email: info@coverity.com

Coverity Inc. Headquarters
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: (800) 873-8193
International Sales: +1 (415) 321-5237
Email: sales@coverity.com